"pydvbcss - a library implementing DVB protocols for Companion Screen Synchronisation

Release 0.3.1-release

British Broadcasting Corporation

February 21, 2016

Contents

1 Run the examples 3

2 DVB CSS Protocol modules 9

3 Clocks, Time and Scheduling modules 35

4 Internal implementation details 57

5 Getting started 63

6 State of implementation 65

7 License and Contributing 67

8 Contact and discuss 69

Python Module Index 71

DVB protocols for synchronisation between TV Devices and Companion Screen Applications.

Release 0.3.1-release

Licence Apache License v2.0.

Latest Source https://github.com/BBC/pydvbcss/tree/master/

How to install https://github.com/BBC/pydvbcss/tree/master/README.md

CHAPTER 1

Run the examples

The code in the *examples* directory demonstrates how to create and control servers and clients for all three protocols: CSS-CII, CSS-TS and CSS-WC.

- ***WallClockServer.py** and **WallClockClient.py***
- ***CIIServer.py** and **CIIClient.py***
- ***TSServer.py** and **TSClient.py***
- ***TVDevice.py***

There are instructions below on how to run the examples and see them interact with each other.

See the sources here: on github

1.1 WallClockServer.py and WallClockClient.py

1.1.1 Get started

The *WallClockServer* and *WallClockClient* examples use the library to implement a simple server and client for the CSS-WC protocol.

First start the server, specifying the host and IP to listen on:

```
$ python examples/WallClockServer.py 127.0.0.1 6677
```

Leave it running in the background and start a client, telling it where to connect to the server:

```
$ python examples/WallClockClient.py 127.0.0.1 6677
```

Note: The wall clock protocol is connectionless (it uses UDP) This means the client will not report an error if you enter the wrong IP address or port number.

Watch the "dispersion" values which indicate how much margin for error there is in the client's wall clock estimate. If the value is very large, this means it is not receiving responses from the server.

1.1.2 How they work

WallClockServer.py [source]

It works by instantiating a `WallClockServer` object and providing that object with a *`clock`* object to be used as the Wall Clock that is to be served.

At the command line you can override default options for the ip address and port the server binds to; the maximum frequency error it reports and whether it sends "follow-up" responses to requests.

Use the `--help` command line option for usage information.

WallClockClient.py [source]

It works by instantiating a `WallClockClient` object and plugs into that object a `LowestDispersionCandidate` algorithm object that adjusts a *`TunableClock`* representing the Wall Clock.

At the command line you must specify the host and port of the Wall Clock server. Default options can be overridden for the IP address and port that the client listens on.

Use the `--help` command line option for usage information.

1.2 CIIServer.py and CIIClient.py

1.2.1 Get started

The *CIIServer* and *CIIClient* examples implement the *CSS-CII* protocol, with the server sharing some pretend CII status information with the client.

First start the server:

```
$ python examples/CIIServer.py
```

The server listens on 127.0.0.1 on port 7681 and accepts WebSocket connections to *ws://<ip>:<port>/cii*.

Leave it running in the background and connect using the client and see how the CII data is pushed by the server whenever it changes:

```
$ python examples/CIIClient.py ws:/127.0.0.1:7681/cii
```

1.2.2 How they work

CIIServer.py [source]

It works by setting up a web server and the ws4py plug-in for cherrypy that provides WebSockets support. It then instantiates a `CIIServer` and mounts it into the cherrypy server at the URL resource path "/cii".

While the server is running, it pretends to be hopping between a few different broadcast channels every 7 seconds, with a 2 second "transitioning" period on each hop.

This is an artificially simple example and does not provide values for most properties of the CII message - such as a MRS URL, or any URLs for a WC or TS endpoints.

It does not do any media presentation, but just provides a CSS-CII server with some pretend data.

This server, by default, serves on port 7681 and provides a CSS-CII service at the URL resource path `/cii`. It can therefore be connected to using the WebSocket URL "ws://<host>:7681/cii" e.g. "ws://127.0.0.1:7681/cii". Command line options can be used to override these defaults and to reduce the amount of logging output.

Use the `--help` command line option for more detailed usage information.

CIIServer.py [source]

It works by instantiating a `CIIClient` and attaching handler functions to be notified of when connection and disconnection occurs and of changes to the CII information being pushed from the server.

At the command line you must specify:

- the WebSocket URL of the CSS-CII server, in the form *ws://<host>:<port>/<path>*

Command line options can be used to reduce the amount of logging output.

Use the `--help` command line option for usage information.

1.3 TSServer.py and TSClient.py

1.3.1 Get started

The *TServer* and *TSClient* examples implement the *CSS-TS* protocol, with the server pretending to have a few different timelines for a DVB broadcast service (where the content ID is a DVB URL).

First start the server:

```
$ python examples/TSServer.py
```

The server listens on 127.0.0.1 on port 7681 and accepts WebSocket connections to *ws://<ip>:<port>/ts*. It also includes a wall clock server, also on 127.0.0.1 on port 6677.

Leave it running in the background and connect using the client and see how the client is able to synchronise and periodically print an estimate of the timeline position (converted to units of seconds):

```
$ python examples/TSClient.py ws://127.0.0.1:7681/ts udp://127.0.0.1:6677 "dvb://" "urn:dvb:css:timel
```

Here we have told it to request a timeline for whatever content the server thinks it is showing provided that the content ID begins with "dvb://". Assuming that matches, then the timeline is to be a PTS timeline, which ticks at 90kHz (the standard rate of PTS in an MPEG transport stream).

1.3.2 How they work

TSServer.py [source]

It works by setting up a web server and the ws4py plug-in for cherrypy that provides WebSockets support. It then instantiates a `TSServer` and mounts it into the cherrypy server at the URL resource path "/ts". It also includes a wall clock server.

It does not play any media, but instead serves an imaginary set of timelines.

It creates *`clock`* objects to represent timelines and the wall clock. `SimpleClockTimelineSource` objects are used to interface the clocks as sources of timelines to the TS server object.

It has a hardcoded DVB URL as the content ID (displayed when you start it running) and provides the following timelines:

- "*urn:dvb:css:timeline:pts*" ... a PTS timeline
- "*urn:dvb:css:timelime:temi:1:1*" ... a TEMI timeline ticking at 1kHz
- "*urn:dvb:css:timelime:temi:1:2*" ... the same, but it takes 10 seconds for the server to begin providing this timeline after a client first requests it
- "*urn:pydvbcss:sporadic*" ... a meaningless timeline whose availability toggles every 10 seconds.

The PTS and TEMI timelines both pause periodically and have their timing tweaked by a fraction of a second. The "sporadic" timeline shows how the protocol supports having timelines appear (become available) and disappear (become unavailable) while a client is connected.

By default, this server serves at 127.0.0.1 on port 7681 and provides a CSS-TS service at the URL *ws://127.0.0.1:7681/ts*. It also provides a wall clock server bound to 0.0.0.0 on UDP port 6677. Command line options can be used to override these defaults and to reduce the amount of logging output.

Use the `--help` command line option for more detailed usage information.

TSClient.py [source]

It works by implementing *both* a wall clock client and a CSS-TS client. A `TSClientClockController` object is instantiated and provided with a `CorrelatedClock` object to represent the synchronisation timeline. The controller adjusts the clock object to match the timeline information coming from the server.

At the command line you must specify:

- the WebSocket URL of the CSS-TS server, in the form *ws://<host>:<port>/<path>*
- a *udp://<host>:<port>* format URL for the Wall Clock server
- The content ID stem and timeline selector to be used when requesting the timeline
- The tick rate of the timeline.

Default options can be overridden for the IP address and port that the Wall Clock client binds to and to reduce the amount of logging output.

Use the `--help` command line option for usage information.

1.4 TVDevice.py

1.4.1 Get started

This is a very simple example of a server running all three protocols (CSS-WC, CSS-TS and CSS-CII). It pretends to be showing a DVB broadcast service and able to provide a PTS or TEMI timeline for it.

First start the server:

```
$ python examples/TVDevice.py
```

While we leave it running in the background, we can try to interact with it using the various example clients described above.

By default it provides a wall clock server on 127.0.0.1 port 6677

```
$ python examples/WallClockClient.py 127.0.0.1 6677
```

... and a CSS-CII server that can be reached at *ws://127.0.0.1:7681/cii*

```
$ python examples/CIIClient.py ws:/127.0.0.1:7681/cii
```

... and a CSS-TS server that can be reached at *ws://127.0.0.1:7681/ts*

```
$ python examples/TSClient.py ws://127.0.0.1:7681/ts udp://127.0.0.1:6677 "dvb://" "urn:dvb:css:timel
```

1.4.2 How it works

TVDevice.py [source]

This example works by setting up a web server and the ws4py plug-in for cherrypy that provides WebSockets support. It then instantiates a `TSServer` and `CIIServer` and mounts it into the cherrypy server. It also includes a wall clock server.

It does not play any media, but instead serves an imaginary set of timelines and pretends to be presenting a broadcast service.

It creates *clock* objects to represent timelines and the wall clock. `SimpleClockTimelineSource` objects are used to interface the clocks as sources of timelines to the TS server object.

It has a hardcoded DVB URL as the content ID (displayed when you start it running) and provides the following timelines:

- *urn:dvb:css:timeline:pts* ... a PTS timeline
- *urn:dvb:css:timelime:temi:1:1* ... a TEMI timeline ticking at 1kHz

The PTS and TEMI timelines both start ticking up from zero the moment the server starts.

By default, this server serves at 127.0.0.1 on port 7681 and provides a CSS-CII service at the URL *ws://127.0.0.1:7681/cii* and a CSS-TS service at the URL *ws://127.0.0.1:7681/ts*. It also provides a wall clock server bound to 0.0.0.0 on UDP port 6677. Command line options can be used to override these defaults and to reduce the amount of logging output.

Use the `--help` command line option for more detailed usage information.

CHAPTER 2

DVB CSS Protocol modules

Module: *dvbcss.protocol*

The `dvbcss.protocol` module contains classes to implement the CSS-CII, CSS-TS and CSS-WC protocols. For each protocol there are objects to represent the messages that flow across the protocols and classes that implement clients and servers for the protocols.

2.1 CSS-CII protocol

2.1.1 CSS-CII Protocol introduction

Here is a quick introduction to the CSS-CII protocol. For full details, refer to the DVB specification ETSI 103 286 part 2.

The CSS-CII protocol is for sharing the server's (e.g. TV's) current "Content Identifier and other Information" (yes really!) with the client (e.g. companion). It also includes the URL of the CSS-TS and CSS-WC servers so the client knows where to find them.

CII comprises a set of defined properties. The server pushes state update messages containing some or all properties (at minimum those that have changed). How often these messages are pushed and which properties are included are up to the server.

It is a WebSockets based protocol and messages are in JSON format.

- *Sequence of interaction*
- *CII message properties*

Sequence of interaction

The client is assumed to already know the WebSocket URL for the CSS-CII server (for example: because the TV chooses to advertise it via a network service discovery mechanism).

1. The client connects to the CSS-TS server. Either this is refused via an HTTP status code response, or it is accepted.

2. The server immediately responds with a first CII state update message. This contains (at minimum) all properties whose values are not null.

3. The server can re-send the CII state update message as often as it wishes. At minimum it will do so when one or more of the properties have changed value. The server will, at minimum, include the properties that have changed, but could also include others in the message.

This protocol is a state update mechanism. The client is locally mirroring the state of the TV by remembering the most recent values received for each of the properties. When a message is received it updates that local state and can react to any changes if it needs to.

Any messages sent by the client are ignored by the server.

CII message properties

Every message sent by the server is a CII message and consists of a single JSON object with zero, one, more or all of the following properties:

- `protocolVersion` - currently "1.1" and must be included in the first message sent by the server after the client connects.
- `contentId` - a URI representing the ID of the content being presented by the server. This will be a variant on a DVB URL ("dvb://") for DVB broadcast services, or the URL of the MPD for MPEG DASH streams.
- `contentIdStatus` - whether the content Id is in its "final" form or whether it is a "partial" version until full information is available. For example: a DVB broadcast content ID might not include some elements until the TV detects certain metadata in the broadcast stream which can take a few seconds.
- `presentationStatus` - Primarily, whether presentation of the content is "okay", "transitioning" from one piece of content to the next, or in a "fault" condition. This can be extended by suffixing space separated additional terms after the primary term.
- `mrsUrl` - The URL of an MRS server.
- `tsUrl` - The WebSockets URL of the CSS-TS server that a client should use if it wants to do Timeline Synchronisation.
- `wcUrl` - The UDP URL ("udp://<host>:<port") of the CSS-WC server.
- `teUrl` - The WebSockets URL of the CSS-TE server that a client should use if it wants to receive Trigger Events.
- `timelines` - a list of zero, one or more timelines that the TV believes to be available for synchronising to.
- `private` - Extension mechanism to carry additional private data.

An example *CII* message:

```
{
        "protocolVersion"    : "1.1",
        "mrsUrl"             : "http://css.bbc.co.uk/dvb/233A/mrs",
        "contentId"          : "dvb://233a.1004.1044;363a~20130218T0915Z--PT00H45M",
        "contentIdStatus"    : "partial",
        "presentationStatus" : "okay",
        "wcUrl"              : "udp://192.168.1.5:5800",
        "tsUrl"              : "ws://192.168.1.8:5815",
        "timelines" : [
                {
                        "timelineSelector"   : "urn:dvb:css:timeline:temi:1:1",
                        "timelineProperties" : {
                                "unitsPerTick"   : 5,
                                "unitsPerSecond" : 10
                        }
                }
        ]
}
```

Another example where the contentId has changed, due to a channel change. The server has chosen to omit properties that have not changed since the previous message:

```
{
        "contentId"          : "dvb://233a.1004.1044;364f~20130218T1000Z--PT01H15M",
        "contentIdStatus"    : "partial",
}
```

2.1.2 CSS-CII Message objects

Module: *dvbcss.protocol.cii*

- *Examples*
- *Classes*
  - *CII Message*
  - *Timeline Option*

A `CII` object represents a CII message sent from server to client via the CSS-CII protocol.

A `TimelineOption` object describes a timeline selector and the tick rate of the timeline if that selector is used to request a timeline from the CSS-TS server. It is carried in a list in the `timelines` property of a `CII` message.

Examples

`CII` messages:

```
>>> from dvbcss.protocol.cii import CII
>>> from dvbcss.protocol import OMIT

>>> jsonCiiMessage = \"""\
...     { "protocolVersion":"1.1",
...       "contentId":"dvb://1234.5678.01ab",
...       "contentIdStatus":"partial"
...     }
... \"""

>>> cii = CII.unpack(jsonCiiMessage)
>>> cii.contentId
'dvb://1234.5678.01ab'

>>> print cii.mrsUrl
OMIT

>>> cii.protocolVersion = OMIT
>>> cii.pack()
'{contentId":"dvb://1234.5678.01ab","contentIdStatus":"partial"}'
```

`TimelineOption` within CII messages:

```
>>> from dvbcss.protocol.cii import CII, TimelineOption

>>> t1 = TimelineOption(timelineSelector="urn:dvb:css:timeline:pts", unitsPerTick=1, unitsPerSecond=9
>>> t2 = TimelineOption(timelineSelector="urn:dvb:css:timeline:temi:1:1", unitsPerTick=1, unitsPerSec
```

```
>>> print t1.timelineSelector, t1.unitsPerTick, t1.unitsPerSecond, t1.accuracy
urn:dvb:css:timeline:pts 1 90000 OMIT
```

```
>>> cii = CII(presentationStatus="final", timelines=[t1, t2])
>>> cii.pack()
'{ "presentationStatus": "final",
   "timelines": [ { "timelineProperties": {"unitsPerSecond": 90000, "unitsPerTick": 1},
                    "timelineSelector": "urn:dvb:css:timeline:pts"
                  },
                  { "timelineProperties": {"unitsPerSecond": 1000, "unitsPerTick": 1},
                    "timelineSelector": "urn:dvb:css:timeline:temi:1:1"
                  }
                ]
 }'
```

Classes

CII Message

class dvbcss.protocol.cii.**CII**(***kwargs*)

Object representing a CII message used in the CSS-CII protocol.

Initialisation takes the following parameters, all of which are optional keyword arguments that default to *OMIT* :

Parameters

- **protocolVersion** (*OMIT* or "1.1") – The protocol version being used by the server.
- **mrsUrl** (*OMIT* or str) – The URL of an MRS server known to the server.
- **contentId** (*OMIT* or str) – Content identifier URI.
- **contentIdStatus** (*OMIT* or "partial" or "final") – Content identifier status.
- **presentationStatus** (*OMIT* or list of str) – Presentation status as a list of one or more strings, e.g. `[ "okay" ]`
- **wcUrl** (*OMIT* or str) – CSS-WC server endpoint URL in the form "udp://<host>:<port>"
- **tsUrl** (*OMIT* or str) – CSS-TS server endpoint WebSocket URL
- **teUrl** (*OMIT* or str) – CSS-TE server endpoint WebSocket URL
- **timelines** (*OMIT* or list of *TimelineOption*) – List of timeline options.
- **private** (*OMIT* or *Signalling that a property is to be omitted from a message*) – Private data.

The attributes of the object have the same name as the CII message properties:

- *protocolVersion*
- *mrsUrl*
- *contentId*
- *contentIdStatus*
- *presentationStatus*

- *wcUrl*
- *tsUrl*
- *teUrl*
- *timelines*
- *private*

Properties are accessed as attributes of this object using the same name as their JSON property name.

Converting to and from JSON representation is performed using the *pack()* method and *unpack()* class method. Properties set to equal *OMIT* will be omitted when the message is packed to a JSON representation.

protocolVersion = OMIT
(read/write *OMIT* or "1.1") The protocol version being used by the server.

mrsUrl = OMIT
(read/write *OMIT* or `str`) The URL of an MRS server known to the server

contentId = OMIT
(read/write *OMIT* or `str`) Content identifier (URL)

contentIdStatus = OMIT
(read/write *OMIT* or "partial" or "final") Content identifier status

presentationStatus = OMIT
(read/write *OMIT* or `list` of `str`) Presentation status, e.g. `[ "okay" ]`

wcUrl = OMIT
(read/write *OMIT* or `str`) CSS-WC server endpoint URL in form "udp://<host>:<port>"

tsUrl = OMIT
(read/write *OMIT* or `str`) CSS-TS server endpoint WebSocket URL

teUrl = OMIT
(read/write *OMIT* or `str`) CSS-TE server endpoint WebSocket URL

timelines = OMIT
(read/write *OMIT* or `list`\`(:class:`TimelineOption`)) Timeline options

private = OMIT
(*OMIT* or `list` of `dict`) Private data as a `list` of `dict` objects that can be converted to JSON by `json.dumps()`. Each dict must contain at least a key called "type" with a URI string as its value.

classmethod allProperties()
Returns a list of all property names, whether OMITted or not

combine(*diff*)
Copies this CII object, and updates that copy with any properties (that are not omitted) in the CII object supplied as the *diff* argument. The updated copy is then returned.

Parameters **diff** – (*CII*) A CII object whose properties (that are not omitted) will be used to update the copy before it is returned.

new = old.combine(diff) is equivalent to the following operations:

```
new = old.copy()
new.update(diff)
```

copy()

Returns a copy of this CII object. The copy is a deep copy.

definedProperties()
Returns a list of the names of properties whose value is not OMIT

classmethod diff(*old*, *new*)
Parameters
- **old** – (*CII*) A CII object
- **new** – (*CII*) A CII object

Returns CII object representing changes from old to new CII objects.

If in the new CII object a property is OMITted, it property won't appear in the returned CII object that represents the changes.

If in the old CII object a property is OMITted, but it has a non-omitted value in the new object, then it is assumed to be a change.

pack()
Returns string containing JSON representation of this message.
Throws ValueError if there are values for properties that are not permitted.

classmethod unpack(*msg*)
Convert JSON string representation of this message encoded as a *CII* object.
Throws ValueError if not possible.

update(*diff*)
Updates this CII object with the values of any properties (that are not omitted) in the CII object provided as the *diff* argument.

Note that this changes this object.
Parameters diff – (*CII*) A CII object whose properties (that are not omitted) will be used to update this CII object.

Timeline Option

class dvbcss.protocol.cii.**TimelineOption**(*timelineSelector*, *unitsPerTick*, *unitsPerSecond*, *accuracy=OMIT*, *private=OMIT*)
Object representing a CSS-CII Timeline Option used in the "timelines" property of a CII message.

Initialisation takes the following parameters:

Parameters

- **timelineSelector** (*str*) – The timeline selector
- **unitsPerTick** (*int*) – Denominator of tick rate (in ticks per second) for the corresponding timeline
- **unitsPerSecond** (*int*) – Numerator of tick rate (in ticks per second) for the corresponding timeline
- **accuracy** (*OMIT* or float) – Optional indication of timeline accuracy
- **private** (*OMIT* or *Signalling that a property is to be omitted from a message*) – Optional private data.

It represents a timeline selector and the tick rate of the timeline if that selector is used to request a timeline from the CSS-TS server. It is carried in a list in the *timelines* property of a *CII* message.

The tick rate of the timeline is expressed by the *unitsPerTick* and *unitsPerSecond* values. The tick rate in ticks per second is equal to *unitsPerTick* / *unitsPerSecond*.

Accuracy and private data are optional, but the other fields are mandatory.

The attributes of the object have the same name as the relevant CII message properties:

- *timelineSelector*
- *unitsPerTick*
- *unitsPerSecond*
- *accuracy*
- *private*

Converting to and from JSON representation is performed using the *pack()* method and *unpack()* class method. Properties set to equal *OMIT* will be omitted when the message is packed to a JSON representation.

timelineSelector = OMIT
(str) The timeline selector

unitsPerTick = OMIT
(int) The units per tick of the timeline

unitsPerSecond = OMIT
(int) The units per second of the timeline

accuracy = OMIT
(*OMIT* or float) The accuracy of the timeline with respect to the content in seconds.

private = OMIT
(*OMIT* or list of dict) Private data as a list of dict objects that can be converted to JSON by json.dumps(). Each dict must contain at least a key called "type" with a URI string as its value.

classmethod decode(*struct*)
Internal method used by a *CII* message object when unpacking to JSON format.

classmethod encode(*item*)
Internal class method used by a *CII* message object when packing to JSON format.

pack()
Returns string containing JSON presentation of this message.

classmethod unpack(*msg*)
Convert JSON string representation of this message encoded as a *TimelineOption* object.
Throws ValueError if not possible.

2.1.3 CSS-CII Clients

Module: *dvbcss.protocol.client.cii*

- *Classes*
  - ***CIIClient***
  - ***CIIClientConnection***

Classes

CIIClient

CIIClientConnection

2.1.4 CSS-CII Servers

Module: *dvbcss.protocol.server.cii*

- *Classes*
  - ***CIIServer** - CII Server handler for cherrypy*

Classes

CIIServer - CII Server handler for cherrypy

This package provides objects for representing messages exchanged via the DVB CSS-CII protocol and for implementing clients and servers.

The CII protocol is a mechanism for sending state updates from server to client. The state of the server can be represented by a `CII` message where every property is populated with a value. The server can send complete CII messages or partial ones containing only the properties that have changed value since the last message. The client must track these changes to maintain its own local up-to-date copy of the complete state.

Modules for using the CSS-CII protocol:

- `dvbcss.protocol.cii` : objects for representing and packing/unpacking the CSS-CII protocol messages.
- `dvbcss.protocol.client.cii` : implementations of a client for a CSS-CII connection.
- `dvbcss.protocol.server.cii` : implementations of a server for a CSS-CII connection.

2.2 CSS-TS protocol

2.2.1 CSS-TS Protocol introduction

Here is a quick introduction to the CSS-TS protocol. For full details, refer to the DVB specification ETSI 103 286 part 2.

The CSS-TS protocol is for *Timeline Synchronisation*. Via this protocol, the server (e.g. TV) pushes timestamps to the client (e.g. companion) to keep it up-to-date on the progress of a particular timeline. The timeline to use is requested by the client at the beginning of the interaction.

A client can also report its own timing and what range of timings it can cope with. This allows the client to negotiate a mutually achievable timing with the server, although the server is under no obligation and can choose to ignore this information.

It is a WebSockets based protocol and messages are in JSON format.

- *Sequence of interaction*
- *Determining timeline selection and availability*
- *What does a timestamp convey?*

Sequence of interaction

The client is assumed to already know the WebSocket URL for the CSS-TS server (usually from the information received via the CSS-CII protocol).

1. The client connects to the CSS-TS server. Either this is refused via an HTTP status code response, or it is accepted.

2. The client then immediately sends an initial *SetupData* message to request the timeline to synchronise with.

3. The server then starts sending back *ControlTimestamp* messages that update the client as to the state of that timeline. This state says either that the timeline is currently unavailable, or that it is available, and here is how to calculate the timeline position from the wall clock position.The server sends as frequently or infrequently as it likes, but will at least send them if there is a meaningful change in the timeline.

4. The client can, optionally, send its own *AptEptLpt* messages to inform the server of what it is doing, and the range of different timings it can achieve for its media (e.g. what is the earliest and latest timings it can achieve). However this is purely informative. A server is not obliged to do anything with this information.

Determining timeline selection and availability

The *SetupData* message conveys to the CSS-TS server details of what timeline the client wants to synchronise to.

The CSS-TS server determines, at any given moment, if a timeline is available by checking if:

1. the stem matches the current content identifier for what is being presented at the server (meaning that the stem matches the left hand most subset of the content id);
2. and the timeline selector identifies a timeline that exists for the content being presented at the server.

While the above is true, the timeline is "available". While it is not true, it is "unavailable". The CSS-TS connection is kept open irrespective of timeline availability. The server indicates changes in availability via the *ControlTimestamp* messages it sends.

Example *SetupData* message; requesting a PTS timeline for a particular DVB broadcast channel, but not being specific about which event (programme in the EPG):

```
{
        "contentIdStem"     : "dvb://233a.1004.1044",
        "timelineSelector" : "urn:dvb:css:timeline:pts"
}
```

What does a timestamp convey?

It represents a relationship between Wall Clock time and the timeline of the content being presented by the TV Device. It is sometimes referred to as a *(point of) correlation* between the wall clock and the timeline.

This relationship can be visualised as a line that maps from wall clock time (on one axis) to timeline time (on the other axis). The (content-time, wall-clock-time) correlation is a point on the line. The timelineSpeedMultiplier represents the slope. The tick rates of each timeline are the units (the scale of each axis).

The CSS-TS server sends *ControlTimestamp* messages to clients, and clients can, optionally, send back `AptEptLpt` messages.

A ControlTimestamp can also tell a client if a timeline is unavailable by having null values for the contentTime and timelineSpeedMultiplier properties. Non-null values mean the timeline is available.

AptEptLpt messages enables a client to inform a server of what time it is presenting at (the "actual" part of the timestamp) and also to indicate the earliest and latest times it could present. It is, in effect, three correlations bundled into one message, to represent each of these three aspects. Earliest and Latest correlations are allowed to have -infinity and +infinity for the wall clock time to indicate that the client has no limits on how early, or late, it can present.

An example *ControlTimestamp* indicating the timeline is unavailable:

```
{
        "contentTime"             : null,
        "wallClockTime"           : "116012000000",
        "timelineSpeedMultiplier" : null
}
```

An example *ControlTimestamp* providing a correlation for an available timeline:

```
{
        "contentTime"             : "834188",
        "wallClockTime"           : "116012000000",
        "timelineSpeedMultiplier" : 1.0
}
```

An example of an *AptEptLpt* message, indicating the current presentation timing being used by the client; a limit on how early it can present; but no limit on how long it can delay (buffer):

```
{
        "actual" : {
                "contentTime"   : "834190",
                "wallClockTime" : "115992000000"
        },
        "earliest" : {
                "contentTime"   : "834190",
                "wallClockTime" : "115984000000"
        },
        "latest" : {
                "contentTime"   : "834190",
                "wallClockTime" : "plusinfinity"
        }
}
```

2.2.2 CSS-TS Message objects

Module: *dvbcss.protocol.ts*

- *Examples*
- *+/- infinity*
- *Classes*
  - *setup-data*
  - *Control Timestamp*
  - *AptEptLpt (Actual, Earliest and Latest Presentation Timestamp)*
  - *Timestamp*

A *SetupData* object represents a setup-data message sent by a client to a server immediately after opening a CSS-TS protocol connection.

A *ControlTimestamp* object represents a Control Timestamp message sent by the server to the client.

A *AptEptLpt* object represents an Actual, Earliest and Latest Presentation Timestamp message that may be sent by a client to the server.

The *Timestamp* objects are used in the above message objects to represent the relationship between wall clock time and content (timeline) time.

Examples

SetupData examples:

```
>>> from dvbcss.protocol.ts import SetupData
>>> from dvbcss.protocol import OMIT

>>> s = SetupData(timelineSelector="urn:dvb:css:timeline:pts", ciStem="dvb://1004")
>>> s.pack()
'{"timelineSelector": "urn:dvb:css:timeline:pts", "contentIdStem": "dvb://1004"}'
```

```
>>> jsonMessage = \"""\
... { "timelineSelector":"urn:dvb:css:timeline:temi:1:1",
...   "contentIdStem":""
... }
... \"""
>>> SetupData.unpack(jsonMessage)
SetupData(ciStem="", timelineSelector="urn:dvb:css:timeline:temi:1:1", private=OMIT)
```

ControlTimestamp examples:

```
>>> from dvbcss.protocol.ts import ControlTimestamp, Timestamp

>>> t = Timestamp(contentTime=12345, wallClockTime=900028432)
>>> ct = ControlTimestamp(timestamp=t, timelineSpeedMultiplier=1)
>>> ct.pack()
'{"timelineSpeedMultiplier": 1.0, "wallClockTime": "900028432", "contentTime": "12345"}'
```

```
>>> jsonMessage = \"""\
... { "contentTime" : "1003847",
...   "wallClockTime" : "348957623498576",
...   "timelineSpeedMultiplier" : 2.0
... }
... \"""
>>> c = ControlTimestamp.unpack(jsonMessage)
>>> c.timestamp.contentTime
1003847
>>> c.timestamp.wallClockTime
348957623498576
>>> c.timelineSpeedMultiplier
2.0
```

Actual, Earliest and Latest Presentation Timestamp examples:

```
>>> from dvbcss.protocol.ts import AptEptLpt, Timestamp

>>> te = Timestamp(contentTime=123465, wallClockTime=float("-inf"))
>>> tl = Timestamp(contentTime=123465, wallClockTime=float("+inf"))
>>> ael = AptEptLpt(earliest=te, latest=tl)
>>> ael.pack()
'{"earliest": {"wallClockTime": "minusinfinity", "contentTime": "123465"}, "latest": {"wallClockTime"
```

```
>>> jsonMessage = \"""\
... { "earliest" : { "contentTime" : "1000", "wallClockTime": "10059237" },
...   "latest"   : { "contentTime" : "1000", "wallClockTime": "19284782" },
...   "actual"   : { "contentTime" : "1005", "wallClockTime": "10947820" }
... }
... \"""
>>> ael=AptEptLpt.unpack(jsonMessage)
>>> ael.actual.contentTime
1005
>>> ael.actual.wallClockTime
10947820
```

+/- infinity

For certain timestamp messages it is permissible to convey a time value that is either plus or minus infinity. Use the python `float` to express these values as follows:

```
>>> float("+inf")
inf

>>> float("-inf")
-inf
```

Classes

setup-data

class dvbcss.protocol.ts.**SetupData**(*contentIdStem*, *timelineSelector*, *private=OMIT*)

Object representing a CSS-TS Setup-Data message.

This carries a content identifier stem and a timeline selector string, and is used, in effect, to request the timeline to be synchronised to via the CSS-TS protocol.

Initialisation takes the following parameters:

Parameters

- **contentIdStem** (*str*) – The content identifier stem.
- **timelineSelector** (*str*) – The timeline selector
- **private** (*OMIT* or *Signalling that a property is to be omitted from a message*) – Optional private data.

The attributes of the object have the same name as the SetupData message properties:

- *contentIdStem*
- *timelineSelector*
- *private*

Converting to and from JSON representation is performed using the *pack()* method and *unpack()* class method. Properties set to equal *OMIT* will be omitted when the message is packed to a JSON representation.

contentIdStem

(read/write `str`) The stem (subset starting from the LHS) of a content identifier

timelineSelector
(read/write `str`) The timeline selector

private = OMIT
(read/write *`OMIT`* or *Signalling that a property is to be omitted from a message*) Optional private data.

pack()
Returns string containing JSON representation of this message.
Throws ValueError if there are values for properties that are not permitted.

classmethod unpack(*msg*)
Convert JSON string representation of this message encoded as a *`SetupData`* object.
Throws ValueError if not possible.

Control Timestamp

class dvbcss.protocol.ts.**ControlTimestamp**(*timestamp*, *timelineSpeedMultiplier*)
Object representing a CSS-TS Control Timestamp message.

Initialisation takes the following parameters:

Parameters

- **timestamp** (`Timestamp`) – carries the *contentTime* and *wallClockTime* properties of the Control Timestamp
- **timelineSpeedMultiplier** (`float` or `None`) – the timeline speed multiplier

The attributes of the object have the following relationship to the message properties:

- *`timestamp`*
- *`timelineSpeedMultiplier`*

Converting to and from JSON representation is performed using the *`pack()`* method and *`unpack()`* class method.

timestamp
(read/write *`Timestamp`*) *`Timestamp`* object representing the contentTime and wallClockTime parts of the timestamp)

timelineSpeedMultiplier
(read/write `float` or `None`) Timeline speed. For example: 1 = normal, 0 = pause, -0.5 = half speed reverse. Use *None* only when the Control Timestamp is supposed to indicate that the timeline is unavailable.

pack()
Returns string containing JSON representation of this message.
Throws ValueError if there are values for properties that are not permitted.

classmethod unpack(*msg*)
Convert JSON string representation of this message encoded as a *`ControlTimestamp`* object.
Throws ValueError if not possible.

AptEptLpt (Actual, Earliest and Latest Presentation Timestamp)

class dvbcss.protocol.ts.**AptEptLpt**(*actual=OMIT*, *earliest=Timestamp(contentTime=0, wallClockTime=-inf)*, *latest=Timestamp(contentTime=0, wallClockTime=inf)*)

Object representing a CSS-TS Actual, Earliest and Latest Presentation Timestamp message.

Initialisation takes the following parameters:

Parameters

- **actual** (*OMIT* or *Timestamp*) – Optional timestamp representing the actual presentation timing
- **earliest** (*OMIT* or *Timestamp*) – Timestamp representing the earliest possible presentation timing
- **latest** (*OMIT* or *Timestamp*) – Timestamp representing the latest possible presentation timing

For the *actual presentation timestamp*, the contentTime and wallClockTime must both be non-null integer values.

For the *earliest presentation timestamp*, the contentTime must be a non-null integer. wallClockTime can be a non-null integer or *plus infinity*

For the *latest presentation timestamp*, the contentTime must be a non-null integer. wallClockTime can be a non-null integer or *minus infinity*

The attributes of the object have the same names as the Actual, Earliest and Latest presentation timestamp message properties:

- *actual*
- *earliest*
- *latest*

Converting to and from JSON representation is performed using the *pack()* method and *unpack()* class method. If values of properties do not meet the requirements described above, then *pack()* will raise `ValueError` exceptions.

actual
(read/write `OMIT` or *Timestamp*) Actual presentation timestamp

earliest
(read/write *Timestamp*) Earliest presentation timestamp. The wallClockTime property must be an `int` or `float("+inf")`. It must not be `float("-inf")`.

latest
(read/write *Timestamp*) Latest presentation timestamp. The wallClockTime property must be an `int` or `float("-inf")`. It must not be `float("+inf")`.

pack()
Returns string containing JSON representation of this message.
Throws ValueError if there are values for properties that are not permitted.

classmethod unpack(*msg*)
Convert JSON string representation of this message encoded as a *AptEptLpt* object.
Throws ValueError if not possible.

Timestamp

This object does not directly represent a message, but is instead used by *ControlTimestamp* and *AptEptLpt* as a representation of a correlation between a content time and a wall clock time.

class dvbcss.protocol.ts.**Timestamp**(*contentTime*, *wallClockTime*)
Object representing a Timestamp part(s) of a *ControlTimestamp* or *AptEptLpt* object.

Initialisation takes the following parameters:

Parameters

- **contentTime** (None or int) – The content time (time on timeline) part of a timestamp
- **wallClockTime** (int or *+/- infinity* float ("+inf") or float ("-inf")) – The wall clock time part of a timestamp

The values for contentTime and wallClockTime are allowed to be arbitrarily large precision integers because they are carried as a string in the JSON representation.

The attributes of the object have the same name as the corresponding message properties:

- *contentTime*
- *wallClockTime*

Converting to and from JSON representation is performed using the pack() method and unpack() class method.

contentTime
(read/write None or large int) The content time part of a timestamp

wallClockTime
(read/write large int or float("+inf") or float("-inf")) The wall clock time part of a timestamp

2.2.3 CSS-TS Clients

Module: *dvbcss.protocol.client.ts*

- *Classes*
  - *TSClientConnection*
  - *TSClientClockController*

Classes

TSClientConnection

TSClientClockController

2.2.4 CSS-TS Servers

Module: *dvbcss.protocol.server.ts*

- *Classes*
  - *TSServer*
  - *TimelineSource*
  - *SimpleTimelineSource*
  - *SimpleClockTimelineSource*
- *Functions*
  - *ciMatchesStem*
  - *isControlTimestampChanged*

Classes

TSServer

TimelineSource

SimpleTimelineSource

SimpleClockTimelineSource

Functions

ciMatchesStem

isControlTimestampChanged

This package provides objects for representing messages exchanged via the DVB CSS-TS protocol and for implementing clients and servers.

The TS protocol is a mechanism for a server to share timeline position and playback speed with a client. In effect it enables a client to synchronise its understanding of the progress of media presentation with that of a server, in terms of a particular timeline.

The client initially sends a *SetupData* message to specify what timeline it wants to synchronise in terms of. The server then periodically sends *ControlTimestamp* messages to inform the client of the state of presentation timing. The client can also send AptEptLpt (Actual, Earliest and Latest Presentation Timestamp) messages to the server to inform it of its playback timing and range of playback timings it can achieve.

The client implementation in this library can control a *CorrelatedClock*, synchronising it to the timeline. The server implementation in this library uses *CorrelatedClock* objects as its source of timelines that it is to share with clients.

Modules for using the CSS-TS protocol:

- *dvbcss.protocol.ts* : objects for representing and packing/unpacking the CSS-TS protocol messages.
- *dvbcss.protocol.client.ts* : implementation of a client for a CSS-TS connection.
- *dvbcss.protocol.server.ts* : implementation of a server for a CSS-TS connection.

2.3 CSS-WC protocol

2.3.1 CSS-WC Protocol Introduction

Here is a quick introduction to the CSS-WC protocol. For full details, refer to the DVB specification ETSI 103 286 part 2.

The CSS-WC protocol is for establishing *Wall clock synchronisation* - meaning that there is a common synchronised sense of time (a "wall clock") between the server (e.g. TV) and client (e.g. companion). This common wall clock is used in the CSS-TS protocol to make it immune to network delays.

The client uses the information carried in the protocol to estimate the server wall clock and attempt to compensate for network latency. This is a connectionless UDP protocol similar to NTP's client-server mode of operation, but much simplified and not intended to set the system real-time clock.

- *Sequence of interaction*
- *Synchronising the wall clock*
- *Message format*

Sequence of interaction

The client is assumed to already know the host and port number of the CSS-WC server (usually from the information received via the CSS-CII protocol).

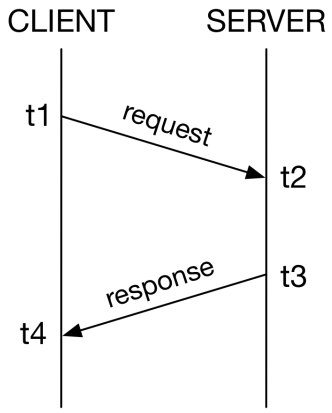
1. The client sends a Wall Clock protocol "request" message to the server.
2. The server sends back a Wall Clock protocol "response" message to the client.

3. If the server is able to more accurately measure when it sent a message *after* it has done so, then it can optionally send a "follow-up response" with this information.

The client repeats this process as often as it needs to.

Synchronising the wall clock

The client notes the time at which the request is sent and the response received, and by the server including the times at which it received the request and sent its response. Using this information the client can estimate the difference between the time of its clock and that of the server. It can also calculate an error bound on this (known as dispersion):



Estimated offset = ((*t3* + *t2*) - (*t4* + *t1*)) / 2

The DVB specification (part 2) contains an annex that goes into more detail on the theory of how to calculate dispersion and how a client can use this as part of a simple algorithm to align its wall clock.

Message format

Requests and responses both have the same fixed 32 byte binary message format. A `WCMessage` carries the following fields:

- Protocol **version** identifier
- Message **type** (request / response / response-before-follow-up / follow-up)
- The **precision** of the server's wall clock
- The **maximum frequency error** of the server's wall clock
- Timevalues (in NTP 64-bit time format, comprising a 32bit word carrying the number of nanoseconds and another 32bit word containing the number of seconds)
  - **Originate timevalue:** when the client sent the request.
  - **Receive timevalue:** when the server received the request.
  - **Transmit timevalue:** when the server sent the response.

The *precision*, *max freq error*, *receive timevalue* and *transmit timevalue* fields only have meaning in a response from a server. Their values do not matter in requests.

2.3.2 CSS-WC Message objects

Module: *dvbcss.protocol.wc*

- *Example usage*
- *Classes*
  - *WCMessage*
  - *Candidate*

The `WCMessage` class represents a CSS-WC protocol message.

The `Candidate` class represents a measurement "candidate" for use by a Wall Clock Client algorithm and is calculated from a `WCMessage` that represents a Wall Clock protocol response message received from a server.

Example usage

Creating a request message at a Wall Clock Client:

```
>>> from dvbcss.protocol.wc import WCMessage
>>> import time

>>> t1 = time.time() * 1000000000

>>> msg = WCMessage(msgtype=WCMessage.TYPE_REQUEST, precision=-10, maxFreqError=256*50, originateNano
>>> packedMessage = msg.pack()
>>> packedMessage
"\x00\x00\xf6\x00\x00\x002\x00TvH'3\xf5\xfc\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
```

Processing a received response message at a Wall Clock Client:

```
>>> from dvbcss.protocol.wc import Candidate

>>> t4 = <nanoseconds-at-which-message-was-received>
>>> msg = WCMessage.unpack(receivedData)
>>> msg.msgtype
1
>>> c = Candidate(msg, t4)
>>> c.rtt
459734573
```

Classes

WCMessage

class dvbcss.protocol.wc.**WCMessage**(*msgtype*, *precision*, *maxFreqError*, *originateNanos*, *receiveNanos*, *transmitNanos*, *originalOriginate=None*)

Create object representing a CSS-WC wall clock request or response message.

Initialisation takes the following parameters:

Parameters

- **msgtype** (*int*) – Type of message. One of: *TYPE_REQUEST*, *TYPE_RESPONSE*, *TYPE_RESPONSE_WITH_FOLLOWUP* and *TYPE_FOLLOWUP*
- **precision** (*int*) – Precision (of the server's wall clock) encoded in log base 2 seconds between -128 and +127 inclusive.
- **maxFreqError** (*int*) – Maximum frequency error (of the server's wall clock) in units of 1/256ths ppm.
- **originateNanos** (*int*) – Originate timevalue in integer number of nanoseconds
- **receiveNanos** (*int*) – Receive timevalue in integer number of nanoseconds
- **transmitNanos** (*int*) – Transmit timevalue in integer number of nanoseconds
- **originalOriginate** (None or (int, int)) – Optional original encoding of the originate timevalue as (seconds, nanos). Overrides *originateNanos* if not *None*.

The originalOriginate parameter, if not None, overrides the originateNanos parameter.

Convert to and from a string containing the binary encoding of this message using the *pack()* method and *unpack()* class method.

msgtype
(read/write int) Type of message. 0=request, 1=response, 2=response-with-followup, 3=followup

precision
(read/write int) Precision encoded in log base 2 seconds between -128 and +127 inclusive. For example: -10 encodes a precision value of roughly 0.001 seconds.

maxFreqError
(read/write int) Maximum frequency error in units of 1/256ths ppm. For example: 12800 encodes a max freq error of 50ppm.

originateNanos
(read/write int) Originate timevalue in integer number of nanoseconds

receiveNanos
(read/write int) Receive timevalue in integer number of nanosecond

transmitNanos
(read/write int) Transmit timevalue in integer number of nanosecond

originalOriginate
(read/write None or (int, int)) Optional original encoding of the originate timevalue as (seconds, nanos). Overrides *originateNanos* when the message is packed if the value is not *None*.

TYPE_FOLLOWUP = 3
Constant: Message type 3 "follow-up response"

TYPE_REQUEST = 0
Constant: Message type 0 "request"

TYPE_RESPONSE = 1
Constant: Message type 1 "response with no follow-up"

TYPE_RESPONSE_WITH_FOLLOWUP = 2
Constant: Message type 2 "response to be followed by a follow-up response"

copy()
Duplicate this wallclock message object

classmethod encodePrecision(*precisionSecs*)
Convert a precision value in seconds to the format used in these messages

getMaxFreqError()
Get frequency error in ppm

getPrecision()
Get precision value in fractions of a second

pack()
Pack wall clock message into binary representation.
Returns String containing the wall clock message in final bitstream form.

setMaxFreqError(*maxFreqErrorPpm*)
Set freq error given a freq error represented as ppm

setPrecision(*precisionSecs*)
Set precision value given a precision represented as factions of a second

classmethod unpack(*data*)
Class method that takes a string containing a wall clock message and unpacks it to a *WCMessage* object.
Parameters **data** (*str*) – String containing binary representation of a Wall Clock message as received from a client or server.
Returns *WCMessage* object representing the wall clock message.

Candidate

class dvbcss.protocol.wc.**Candidate**(*msg*, *nanosRx*)
This object represents a measurement "candidate" to be fed into a Wall Clock Client's algorithm. It is calculated from a *WCMessage* received as a response from a Wall Clock server.

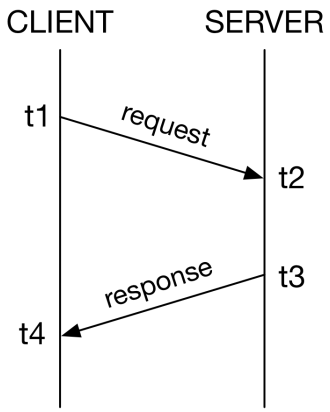
Initialisation takes the following parameters:

Parameters

- **msg** (*WCMessage*) – Response message received from server
- **nanosRx** (*int*) – the time, in nanoseconds, at which it was received (from the server)

Pass in a received WallClockMessage that is a response and this will represent the candidate data derived from that request-response interaction.

Populates properties of this objects with the candidate information. *t1*, *t2*, *t3* and *t4* represent the times of message sending and receiving as shown below:



It also calculates and provides, as properties, the round-trip time (*rtt*) and clock offset estimate (*offset*) based on this measurement.

By default the data is preserved in nanoseconds. However you may use the *toTicks()* method to create a new version of the Candidate object using units of ticks matching a *clock* object you provide.

The exceptions are precision which is measured in seconds, and maximum frequency error which is measured in ppm

t1 = msg.originateNanos
(read only) The time "t1" at which the request was sent in the request-response measurement (nanoseconds, or clock ticks)

t2 = msg.receiveNanos
(read only) The time "t2" at which the request was received in the request-response measurement (nanoseconds, or clock ticks)

t3 = msg.transmitNanos
(read only) The time "t3" at which the response was sent in the request-response measurement (nanoseconds, or clock ticks)

t4 = nanosRx
(read only) The time "t4" at which the response was received in the request-response measurement (nanoseconds, or clock ticks)

offset = ((t3+t2)-(t4+t1))/2
(read only) Server<->client clock offset (nanoseconds, or clock ticks)

rtt = (t4-t1)-(t3-t2)
(read only) Round trip time (nanoseconds, or clock ticks)

isNanos = True
(read only bool) True if this candidate object is in units of nanoseconds for t1, t2, t3, t4, offset and rtt

precision = msg.getPrecision()
(read only) The precision reported by the server in its response (units of factions of a second)

maxFreqError = msg.getMaxFreqError()
(read only) The maximum frequency error reported by the server in its response (units of ppm)

msg = WCMessage
(read only *WCMessage*) The response message from which this candidate was derived

toTicks(*clock*)
Returns a new Candidate object the same as this one but whose measurements have been converted to match the timescale of a clock.

t1, *t2*, *t3*, *t4*, rtt and *offset* of the returned object are converted to units of ticks matching the clock. But *precision* and *maxFreqError* remain unchanged.

Parameters **clock** (dvbcss.clock) – Clock whose *nanosToTicks()* function will be used to create the new candidate object

Returns a copy of this candidate where the units have been converted to 'ticks' according to the tick rate of the supplied clock.

2.3.3 CSS-WC Clients

Modules: *dvbcss.protocol.client.wc* | *dvbcss.protocol.client.wc.algorithm*

- *Algorithms*
- *Functions*
  - *Filter and Prediction algorithm creator*
- *Classes*
  - *WallClockClient*
  - *Dispersion algorithm*
  - *Most recent measurement algorithm*
  - *Filter and Prediction composable algorithms*
    - *Filters*
    - *Predictors*
  - *General helper classes*
    - *Dispersion calculator*
    - *Candidate quality calculator*

Algorithms

Functions

Filter and Prediction algorithm creator

Classes

WallClockClient

Dispersion algorithm

Most recent measurement algorithm

Filter and Prediction composable algorithms

Filters

Predictors

General helper classes

Dispersion calculator

Candidate quality calculator This function is used internally by the `WallClockClient` class. .. autofunction:: dvbcss.protocol.client.wc.algorithm.calcQuality

2.3.4 CSS-WC Servers

- *Classes*
  - *WallClockServer*

Modules: *dvbcss.protocol.server.wc*

- *Classes*
  - *WallClockServer*

Classes

WallClockServer

This package provides objects for representing messages exchanged via the DVB CSS-WC protocol and for implementing clients and servers.

The WC protocol is a simple UDP request response-protocol that enables simple clock synchronisation algorithms to be used to establish a common *wall clock* between a server and client.

There is a `WallClockServer` class providing a self contained Wall Clock server. The `WallClockClient` is designed to allow different algorithms to be plugged in for acting on the results of the request-response interaction to adjust a local *`clock`* object to match the Wall Clock of the server.

Modules for using the CSS-WC protocol:

- *`dvbcss.protocol.wc`* : objects for representing and packing/unpacking the protocol messages.
- *`dvbcss.protocol.client.wc`* : implementation of a client for a CSS-WC connection.
- *`dvbcss.protocol.client.wc.algorithm`* : algorithms to be used with a CSS-WC client.
- *`dvbcss.protocol.server.wc`* : implementation of a server for a CSS-WC connection.

See Protocol server implementation details for information on how the servers are implemented.

2.4 Common types and objects

2.4.1 Signalling that a property is to be omitted from a message

`dvbcss.protocol.`**`OMIT`** **object**

When this object is assigned to an attribute of a protocol message object this indicates that the corresponding property is not included in the JSON representation of that message (it is omitted).

Here is an example. By default nearly all properties of a freshly created CII message object are 'OMIT':

```
>>> from dvbcss.protocol.cii import CII
>>> from dvbcss.protocol import OMIT

>>> c=CII()
>>> print repr(c.contentId)
OMIT

>>> c.wcUrl = "udp://192.168.1.1:6677"
>>> print repr(c.wcUrl)
'udp://192.168.1.1:6677'

>>> c.wcUrl = OMIT
>>> print repr(c.wcUrl)
OMIT
```

2.4.2 Private data

Some protocol messages contain optional properties to carry private data.

Private data is encoded in message objects here as a `list` of `dict` objects where each has a key "type" whose value is a URI.

Each dict can contain any other keys and values you wish so long as they can be parsed by the python `json` module's encoder. For example:

```
example_private_data = [
    { "type" : "urn:bbc.co.uk:pid", "pid":"b00291843",
      "entity":"episode"
    },
    { "type" : "tag:bbc.co.uk/programmes/clips/link-url",
      "http://www.bbc.co.uk/programmes/b1290532/"
```

```
    }
]
```

2.4.3 Exceptions

CHAPTER 3

Clocks, Time and Scheduling modules

This library contains a range of tools for dealing with timing, clocks, and timelines and scheduling code to run at set times.

Contents:

3.1 Montonic time functions

Module: *dvbcss.monotonic_time*

- *Example use*
- *Operating system implementation details*
  - *Windows*
  - *Mac OS X*
  - *Linux*
- *Functions*
  - *time(), timeMicros() and timeNanos()*
  - *sleep()*

This module implements operating system specific access to high resolution monotonic system timers for the following operating systems:

- Windows 2000 Pro or later
- Linux
- Mac OS X

It implements a `time()` function and `sleep()` function that work the same as the `time.time()` and `time.sleep()` functions in the python standard library.

It also adds a `timeNanos()` and `timeMicros()` variants that report time in units of nanoseconds or microseconds instead of seconds.

See operating system specific implementation details below to understand the limitations of the functions provided in this module.

Note: For all supported operating systems, the sleep() function is not guaranteed to use the same underlying timer as the time() and therefore should be considered inaccurate.

3.1.1 Example use

```
>>> import dvbcss.monotonic_time as monotonic_time
>>> monotonic_time.time()
4695.582637038
>>> monotonic_time.timeNanos()
4700164952506L
>>> monotonic_time.timeMicros()
4703471405L
>>> monotonic_time.sleep(0.5)    # sleep 1/2 second
```

3.1.2 Operating system implementation details

The precision and accuracy of the clocks and sleep functions are dependent on the host operating system and hardware. This module can therefore provide no performance guarantees.

Windows

Windows NT 5.0 (Windows 2000 Professional) or later is supported, including the cygwin environment.

The *time()* function and its variants are based on the QueryPerformanceCounter() high resolution timer system call. This clock is guaranteed to be monotonic and have 1 microsecond precision or better.

The *sleep()* function is based on the CreateWaitableTimer() and SetWaitableTimer() system calls.

Note that the *sleep()* function for Windows is not guaranteed to be accurate because it is not possible to create blocking (non polling) delays based on the clock source used. However it should have significantly higher precision than the standard 15ms windows timers and will be fine for short delays.

Mac OS X

The *time()* function and its variants are based on the mach_absolute_time() system call.

The clock is guaranteed to be monotonic. Apple provides no guarantees on precision, however in practice it is usually based on hardware tick counters in the processor or support chips and so is extremely high precision (microseconds or better).

The *sleep()* function is based on the nanosleep() system call. It is unclear whether this uses the same underlying counter as *mach_absolute_time()*.

Linux

The *time()* function and its variants are based on the clock_gettime() system call requesting *CLOCK_MONOTONIC*.

The *sleep()* function is based on the nanosleep() system call. It is unclear whether this uses the same underlying counter as *CLOCK_MONOTONIC*.

3.1.3 Functions

time(), timeMicros() and timeNanos()

dvbcss.monotonic_time.**time**()
Return monotonic time in seconds and fractions of seconds (as a float). The precision is operating system dependent.

dvbcss.monotonic_time.**timeMicros**()
Return monotonic time in integer microseconds. The precision is operating system dependent.

dvbcss.monotonic_time.**timeNanos**()
Return monotonic time in integer nanoseconds. The precision is operating system dependent.

sleep()

dvbcss.monotonic_time.**sleep**(*t*)
Sleep for specified number of second and fractions of seconds (as a float). The precision is operating system dependent.

Throws TimeoutError if the underlying system call used to sleep reported a timeout (OS dependent behaviour)

Throws InterruptedException if a signal or other interruption is received while sleeping (OS dependent behaviour)

Note: For all supported operating systems, the sleep() function is not guaranteed to use the same underlying timer as the time() and therefore should be considered inaccurate.

3.2 Synthesised clocks (dvbcss.clock)

Module: *dvbcss.clock*

- *Introduction*
- *Limitations on tick resolution (precision)*
- *Timers and Sleep functions*
- *Usage examples*
  - *Simple hierarchies of clocks*
  - *Clock speed adjustment*
  - *Translating tick values between clocks*
  - *Implementing new clocks*
- *Functions*
  - ***measurePrecision*** *- estimate measurement precision of a clock*
- *Classes*
  - ***ClockBase*** *- base class for clocks*
  - ***SysClock*** *- Clock based on time module*
  - ***CorrelatedClock*** *- Clock correlated to another clock*
  - ***TunableClock*** *- Clock with dynamically adjustable frequency and tick offset*
  - ***RangeCorrelatedClock*** *- Clock correlated to another clock*

The dvbcss.clock module provides software synthesised clock objects that can be chained together into dependent hierarchies. Use in conjunction with *dvbcss.task* to sleep and run code at set times on these clocks.

3.2.1 Introduction

The classes in this module implement software synthesised clocks from which you can query a current time value. A clock counts in whole numbers of ticks (unlike the standard library `time.time()` function which counts in seconds and fractions of a second) and has a tick rate (expresed in ticks per second).

To use clocks as timers or to schedule code to run later, you must use them with the functions of the *dvbcss.task* module.

To use clocks begin with a root clock that is based on a underlying timing source. The following root clocks are provided:

- *SysClock* **is an root clock based on the** *dvbcss.monotonic_time.time()* **function as the underlying time source**

Other dependent clocks can then be created that have the underlying clock as their parent. and further dependent clocks can be created with those dependents as their parents, creating chains of clocks, each based on their parent and leading back to an underlying clock. The following dependent clocks are provided:

- *CorrelatedClock* **is a fixed tick rate clock where you define the point of correlation between it and its parent.**
- *TunableClock* **is a clock that can have its tick count tweaked and its frequency slewed on the fly.**
- `RangleCorrelatedClock` **is a clock where the relationship to the parent is determined from two points of correlation.**

Dependent clocks can have a different tick rate to their parent and count their ticks from a different starting point.

Dependent clocks allow their relationship to its parent clock to be changed dynamically - e.g. the absolute value, or the rate at which the clock ticks. If your code needs to be notified of such a change, it can bind itself as a dependent to that clock.

The base *ClockBase* class defines the set of methods common to all clocks (both underlying and dependent)

3.2.2 Limitations on tick resolution (precision)

The Clock objects here do not run a thread that counts individual ticks. Instead, to determine the current tick value they query the parent clock for its current tick value and then calculate what the tick value should be.

A clock is therefore **limited in the precision and resolution of its tick value by its parents.** *SysClock*, for example, is limited by the resolution of the underlying time source provided to it by *dvbcss.monotonic_time* module's *dvbcss.monotonic_time.time()* function. And this will be operating system dependent. SysClock also outputs ticks as integer values.

If a parent of a clock only reports whole number (integer) tick values then that also limits the resolution of any clocks that depend on it. For example, a clock that counts in integer ticks only at 25 ticks per second will cause a clock descended from it, with a tick rate of 100 ticks per second, to report tick values that increment 4 ticks at a time ... or worse if a parent of both has an even lower tick rate.

With the clocks provided by this module, only *SysClock* limits itself to integer numbers of ticks. *CorrelatedClock* and *TunableClock* are capable of fractional numbers of ticks provided that the parameters provided to them (e.g. the tickRate) are passed as floating point values (this will force python to do the maths in floating point instead of integer maths).

3.2.3 Timers and Sleep functions

Use the functions of the *dvbcss.task* in conjunction with Clock objects to create code that sleeps, or which triggers callbacks, based on time as measured by these clocks.

3.2.4 Usage examples

Simple hierarchies of clocks

Here is a simple example where a clock represents a timeline and another represents a timeline related to the first by a correlation:

```
from dvbcss.clock import SysClock
from dvbcss.clock import CorrelatedClock

# create a clock based on dvbcss.monotonic_time.time() that ticks in milliseconds
sysClock = SysClock(tickRate=1000)

# create a clock to represent a timeline
baseTimeline = CorrelatedClock(parentClock=sysClock, tickRate=25, correlation=(0,0))

# create a clock representing another timeline, where time zero corresponds to time 100
# on the parent timeline
subTimeline = CorrelatedClock(parentClock=baseTimeline, tickRate=25, correlation=(100,0)
```

At some point later in time during the program, we query the values of all the clocks, confirming that the sub timeline is always 100 ticks ahead of the base timeline.

```
def printTimes():
    sys  = sysClock.ticks()
    base = baseTimeline.ticks()
    sub  = subTimeline.ticks()
    print "SysClock      ticks = %d", sys
    print "Base timeline ticks = %d", base
    print "Sub timeline  ticks = %d", sub

>>> printTimes()
SysClock      ticks = 20000
Base timeline ticks = 500
Sub timeline  ticks = 600
```

Note that in these examples, for clarity, the tick count on the sysClock is artificially low. It would likely be a much larger value.

We then change the correlation for the base timeline, declaring tick 25 on its baseline to correspond to tick 0 on its parent timeline, and both the base timeline and the sub timeline reflect this:

```
>>> baseTimeline.correlation = (0,25)
>>> printTimes()
SysClock      ticks = 30000
Base timeline ticks = 775
Sub timeline  ticks = 875
```

Clock speed adjustment

All clocks have a `speed` property. Normally this is 1.0. Some clock classes support changing this value. This scales the rate at which the clock ticks relative to its parent. For example, 0.5 corresponds to half speed; 2.0 = double speed,

0 = frozen and -1.0 = reverse.

Clocks will take speed into account when returning their current tick position or converting it to or from the tick value of a parent clock. However it does not alter the tickRate property. A child clock will similarly ignore the speed property of a parent clock. In this way, the speed property can be used to tweak the speed of time, or to emulate speed control (fast forward, rewind, pause) for a media timeline.

Here is an example where we create 3 clocks in a chain and all tick initially at 100 ticks per second:

```
>>> import time
>>> baseClock = Sysclock(tickRate=100)
>>> clock1 = TunableClock(parent=baseClock, tickRate=100)
>>> clock2 = TunableClock(parent=clock1, tickRate=100)
```

We confirm that both clock1 and its child - clock2 - tick at 100 ticks per second:

```
>>> print clock1.ticks; time.sleep(1.0); print clock1.ticks
5023
5123
>>> print clock2.ticks; time.sleep(1.0); print clock2.ticks
2150
2250
```

If we change the tick rate of clock1 this affects clock1, but its child - clock2 - continues to tick at 100 ticks every second:

```
>>> clock1.tickRate = 200
>>> print clock1.ticks; time.sleep(1.0); print clock1.ticks
5440
5640
>>> print clock2.ticks; time.sleep(1.0); print clock2.ticks
4103
4203
```

But if we *instead* change the speed multiplier of clock1 then this not only affects the ticking rate of clock1 but also of its child - clock2:

```
>>> clock1.tickRate = 100
>>> clock1.speed = 2.0
>>> print clock1.ticks; time.sleep(1.0); print clock1.ticks
5740
5940
>>> print clock2.ticks; time.sleep(1.0); print clock2.ticks
4603
4803
```

Translating tick values between clocks

The clock classes provide mechanisms to translate a tick value from one clock to a tick value of another clock such that it still represents the same moment in time. So long as both clocks share a common ancestor clock, the conversion will be possible.

toParentTicks() and *fromParentTicks()* converts tick values for a clock to/from its parent clock. *toOtherClockTicks()* will convert a tick value for this clock to the corresponding tick value for any other clock with a common ancestor to this one.

```
from dvbcss.clock import SysClock
from dvbcss.clock import CorrelatedClock
```

```
# create a clock based on dvbcss.monotonic_time.time() that ticks in milliseconds
sysClock = SysClock(tickRate=1000)

#                                       +------------+
#                                    .-- | mediaClock |
# +----------+     +-----------+ <--'   +------------+
# | sysClock | <-- | wallClock |
# +----------+     +-----------+ <--.   +-----------------+
#                                    '-- | otherMediaClock |
#                                        +-----------------+

wallClock = CorrelatedClock(parentClock=sysClock, tickRate=1000000000, correlation=(0,0))
mediaClock = CorrelatedClock(parentClock=wallClock, tickRate=25, correlation=(500021256, 0))
otherMediaClock = CorrelatedClock(parentClock=wallClock, tickRate=30, correlation=(21093757, 0))

# calculate wall clock time 'w' corresponding to a mediaClock time 1582:
t = 1582
w = mediaClock.toParentTicks(t)
print "When MediaClock ticks =", t, " wall clock ticks =", w

# calculate mediaClock time 'm' corresponding to wall clock time 1920395
w = 1920395
m = mediaClock.fromParentTicks(w)
print "When wall clock ticks =", w, " media clock ticks =", m

# calculate other media clock time corresponding to media clock time 2248
t = 2248
o = mediaClock.toOtherClockTicks(otherMediaClock, t)
print "When MediaClock ticks =", t, " other media clock ticks =", o
```

Implementing new clocks

Implement a new clock class by subclassing *ClockBase* and implementing the stub methods.

For example, here is a clock that is the same as its parent (same tick rate) except that its current tick value differs by a fixed offset.

```
from dvbcss.clock import ClockBase

class FixedTicksOffsetClock(ClockBase):

    def __init__(self, parent, offset):
        super(FixedTicksOffsetClock,self).__init__()
        self._parent = parent
        self._offset = offset

    def calcWhen(self, ticksWhen):
        return self._parent.calcWhen(ticksWhen - self._offset)

    def fromParentTicks(self, ticks):
        return ticks + self._offset

    def getParent(self):
        return self._parent

    @property
    def tickRate(self):
```

```
        return self._parent.tickRate

    @property
    def ticks(self)
        return self._parent.ticks + self._offset

    def toParentTicks(self, ticks):
        return ticks - self._offset
```

In use:

```
>>> f = FixedTicksOffsetClock(parentClock, 100)
>>> print parentClock.ticks, f.ticks
216 316
```

When doing this, you must decide whether to allow the speed to be adjusted. If you do, then it must be taken into account in the calculations for the methods: `calcWhen()`, `fromParentTicks()` and `toParentTicks()`.

3.2.5 Functions

measurePrecision - estimate measurement precision of a clock

dvbcss.clock.**measurePrecision**(*clock*, *sampleSize=10000*)

Do a very rough experiment to work out the precision of the provided clock.

Works by empirically looking for the smallest observable difference in the tick count.

Parameters

- **clock** – (:class:dvbcss.clock.ClockBase) Clock to measure
- **sampleSize** – (int) Number of iterations (sample size) to estimate the precision over

Returns (float) estimate of clock measurement precision (in seconds)

3.2.6 Classes

ClockBase - base class for clocks

class dvbcss.clock.**ClockBase**(***kwargs*)

Base class for all clock classes.

By default, adjusting tickRate and speed are not permitted unless a subclass overrides and implements a property setter.

bind(*dependent*)

Bind for notification if this clock changes.

Parameters **dependent** – When this clock changes, the *notify()* method of this dependent will be called, passing a single argument that is this clock.

calcWhen(*ticksWhen*)

Return "when" in terms of the underlying clock behind the root clock implementation (e.g. `monotonic_time.time()` in the case of *SysClock*)

This is a stub for this method. Sub-classes should implement it.

fromParentTicks(*ticks*)
This is a stub for this method. Sub-classes should implement it.

Method to convert from a tick value for this clock's parent to the equivalent tick value (representing the same point in time) for this clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

Returns The specified tick value for the parent clock converted to the timescale of this clock.

Throws StopIteration if this clock has no parent

getEffectiveSpeed()
Returns the 'effective speed' of this clock.

This is equal to multiplying together the speed properties of this clock and all of the parents up to the root clock.

getParent()
This is a stub for this method. Sub-classes should implement it.

Returns *ClockBase* representing the immediate parent of this clock, or None if it is a root clock.

nanos
(read only) The tick count of this clock, but converted to units of nanoseconds, based on the current tick rate (but ignoring the *speed* property).

nanosToTicks(*nanos*)
Convert the supplied nanosecond to number of ticks given the current tick rate of this clock (but ignoring the *speed* property).

Parameters **nanos** – nanoseconds value

Returns number of ticks equivalent to the supplied nanosecond value

notify(*cause*)
Call to notify this clock that the clock on which it is based (its parent) has changed relative to the underlying timing source.

Parameters **cause** – The clock that is calling this method.

Will notify all dependents of this clock (entities that have registered themselves by calling *bind()*).

speed
(read/write `float`) The speed at which the clock is running. Does not change the reported tickRate value, but will affect how ticks are calculated from parent clock ticks. Default = 1.0 = normal tick rate.

tickRate
(read only)The tick rate (in ticks per second) of this clock.

This is a stub for this method. Sub-classes should implement it.

ticks
(read only) The tick count for this clock.

This is a stub for this method. Sub-classes should implement it.

toOtherClockTicks(*otherClock*, *ticks*)
Converts a tick value for this clock into a tick value corresponding to the timescale of another clock.

Parameters

- **otherClock** – A *clock* object representing another clock.

- **ticks** – A time (tick value) for this clock

Returns The tick value of the *otherClock* that represents the same moment in time.

Throws NoCommonClock if there is no common ancestor clock (meaning it is not possible to convert

toParentTicks(*ticks*)
This is a stub for this method. Sub-classes should implement it.

Method to convert from a tick value for this clock to the equivalent tick value (representing the same point in time) for the parent clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

Returns The specified tick value of this clock converted to the timescale of the parent clock.

Throws StopIteration if this clock has no parent

unbind(*dependent*)
Unbind from notification if this clock changes.

Parameters dependent – The dependent to unbind from receiving notifications.

SysClock - Clock based on time module

class dvbcss.clock.**SysClock**(*tickRate=1000000*, ***kwargs*)
A clock based directly on the standard library timer function `monotonic_time.time()`. Returns *integer* ticks when its *ticks* property is queried.

The default tick rate is 1 million ticks per second, but a different tick rate can be chosen during initialisation.

It is not permitted to change the *tickRate* or *speed* property of this clock because it directly represents a system clock.

Parameters tickRate – (int) tick rate for this clock (in ticks per second)

bind(*dependent*)
Bind for notification if this clock changes.

Parameters dependent – When this clock changes, the *notify()* method of this dependent will be called, passing a single argument that is this clock.

calcWhen(*ticksWhen*)

Return "when" in terms of the underlying clock behind the root clock implementation (e.g. `monotonic_time.time()` in the case of *SysClock*)

(Documentation inherited from *ClockBase*)

fromParentTicks(*ticks*)

Method to convert from a tick value for this clock's parent to the equivalent tick value (representing the same point in time) for this clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

returns The specified tick value for the parent clock converted to the timescale of this clock.

throws StopIteration if this clock has no parent

(Documentation inherited from *ClockBase*)

getEffectiveSpeed()
Returns the 'effective speed' of this clock.

This is equal to multiplying together the speed properties of this clock and all of the parents up to the root clock.

getParent()

returns *ClockBase* representing the immediate parent of this clock, or None if it is a root clock.

(Documentation inherited from *ClockBase*)

nanos
(read only) The tick count of this clock, but converted to units of nanoseconds, based on the current tick rate (but ignoring the *speed* property).

nanosToTicks(*nanos*)
Convert the supplied nanosecond to number of ticks given the current tick rate of this clock (but ignoring the *speed* property).

Parameters **nanos** – nanoseconds value

Returns number of ticks equivalent to the supplied nanosecond value

notify(*cause*)
Call to notify this clock that the clock on which it is based (its parent) has changed relative to the underlying timing source.

Parameters **cause** – The clock that is calling this method.

Will notify all dependents of this clock (entities that have registered themselves by calling *bind()*).

speed
(read/write `float`) The speed at which the clock is running. Does not change the reported tickRate value, but will affect how ticks are calculated from parent clock ticks. Default = 1.0 = normal tick rate.

tickRate
(read only)The tick rate (in ticks per second) of this clock.

(Documentation inherited from *ClockBase*)

ticks
(read only) The tick count for this clock.

(Documentation inherited from *ClockBase*)

toOtherClockTicks(*otherClock*, *ticks*)
Converts a tick value for this clock into a tick value corresponding to the timescale of another clock.

Parameters

- **otherClock** – A *clock* object representing another clock.
- **ticks** – A time (tick value) for this clock

Returns The tick value of the *otherClock* that represents the same moment in time.

Throws NoCommonClock if there is no common ancestor clock (meaning it is not possible to convert

toParentTicks(*ticks*)

Method to convert from a tick value for this clock to the equivalent tick value (representing the same point in time) for the parent clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

returns The specified tick value of this clock converted to the timescale of the parent clock.

throws StopIteration if this clock has no parent

(Documentation inherited from *ClockBase*)

unbind(*dependent*)
Unbind from notification if this clock changes.

Parameters **dependent** – The dependent to unbind from receiving notifications.

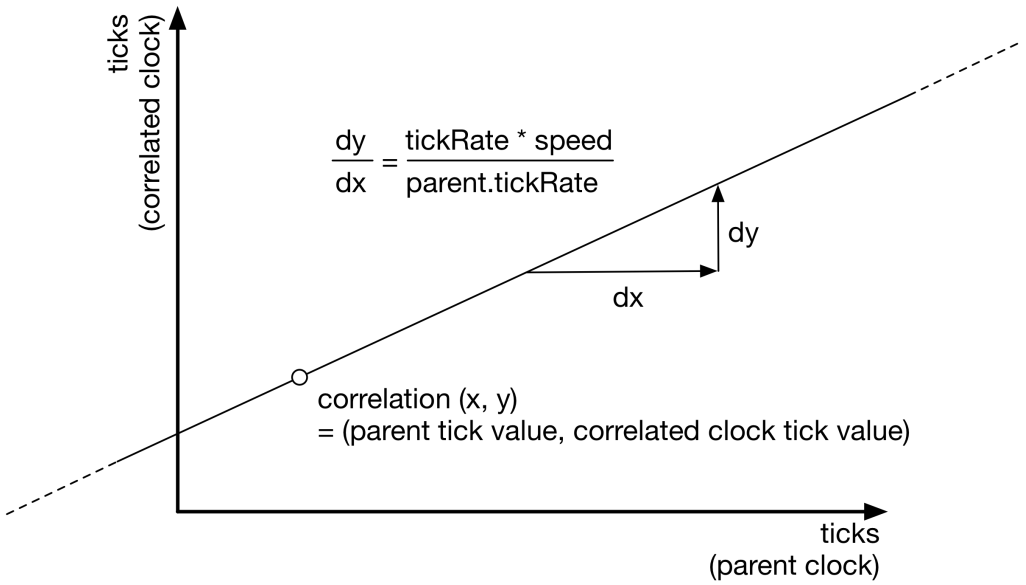
CorrelatedClock - Clock correlated to another clock

class dvbcss.clock.**CorrelatedClock**(*parentClock*, *tickRate*, *correlation=(0, 0)*, ***kwargs*)
A clock locked to the tick count of the parent clock by a correlation and frequency setting.

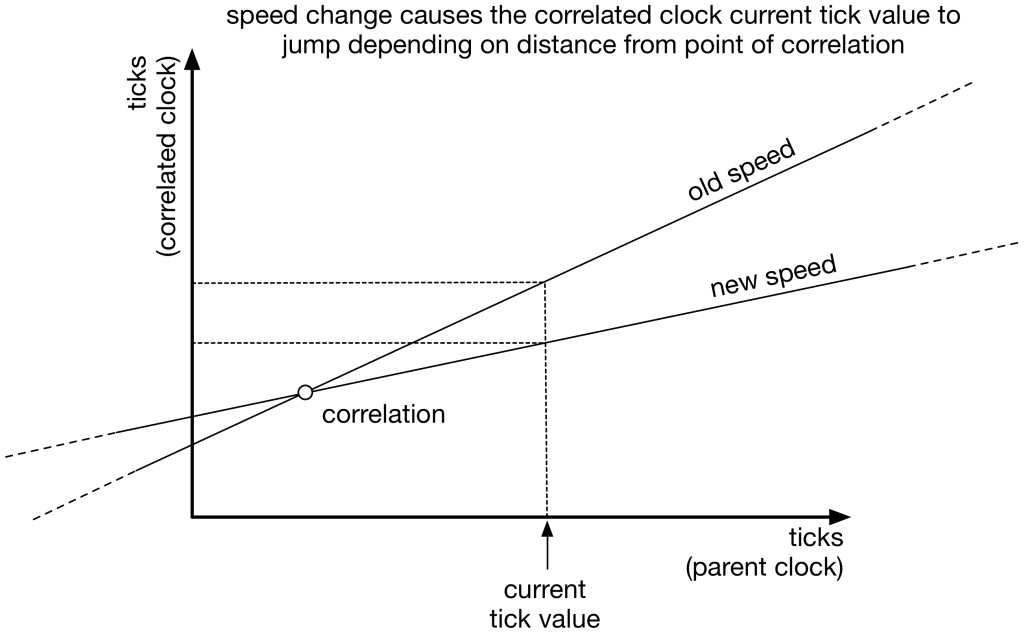
Correlation is a tuple *(parentTicks, selfTicks)*

When the parent clock ticks property has the value *parentTicks*, the ticks property of this clock shall have the value *selfTicks*.

This relationship can be illustrated as follows:



You can alter the correlation and tickRate and speed of this clock dynamically. Changes to tickRate and speed will not shift the point of correlation. This means that a change in tickRate or speed will probably cause the current tick value of the clock to jump. The amount it jumps by will be proportional to the distance the current time is from the point of correlation:



If you want a speed change to only affect the ticks from a particular point (e.g. the current tick value) onwards then you must re-base the correlation. There is a function provided to do that in some circumstances:

```
c = CorrelatedClock(parentClock=parent, tickRate=1000, correlation=(50,78))

   ... time passes ...

# now freeze the clock AT ITS CURRENT TICK VALUE

c.rebaseCorrelationAtTicks(c.ticks)
c.speed = 0

# now resume the clock but at half speed, but again without the tick value jumping
c.correlation = ( parent.ticks, c.ticks )
c.speed = 0.5
```

Note: The maths to calculate and convert tick values will be performed, by default, as integer maths unless the parameters controlling the clock (tickRate etc) are floating point, or the ticks property of the parent clock supplies floating point values.

Parameters

- **parentClock** – The parent clock for this clock.
- **tickRate** – (int) tick rate for this clock (in ticks per second)
- **correlation** – (tuple(int, int)) The intial correlation for this clock. A tuple (parent tick value, this clock tick value)

bind(*dependent*)

Bind for notification if this clock changes.

Parameters **dependent** – When this clock changes, the *notify()* method of this dependent will be called, passing a single argument that is this clock.

calcWhen(*ticksWhen*)

Return "when" in terms of the underlying clock behind the root clock implementation (e.g. `monotonic_time.time()` in the case of *SysClock*)

(Documentation inherited from *ClockBase*)

correlation
Read or change the correlation tuple *(parentTicks, selfTicks)* of this clock to its parent clock.

Assign a new tuple *(parentTicks, selfTicks)* to change the correlation. This value must be a tuple, not a list.

fromParentTicks(*ticks*)

Method to convert from a tick value for this clock's parent to the equivalent tick value (representing the same point in time) for this clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

returns The specified tick value for the parent clock converted to the timescale of this clock.

throws StopIteration if this clock has no parent

(Documentation inherited from *ClockBase*)

getEffectiveSpeed()
Returns the 'effective speed' of this clock.

This is equal to multiplying together the speed properties of this clock and all of the parents up to the root clock.

getParent()

returns *ClockBase* representing the immediate parent of this clock, or None if it is a root clock.

(Documentation inherited from *ClockBase*)

nanos
(read only) The tick count of this clock, but converted to units of nanoseconds, based on the current tick rate (but ignoring the *speed* property).

nanosToTicks(*nanos*)
Convert the supplied nanosecond to number of ticks given the current tick rate of this clock (but ignoring the *speed* property).

Parameters **nanos** – nanoseconds value

Returns number of ticks equivalent to the supplied nanosecond value

notify(*cause*)
Call to notify this clock that the clock on which it is based (its parent) has changed relative to the underlying timing source.

Parameters **cause** – The clock that is calling this method.

Will notify all dependents of this clock (entities that have registered themselves by calling *bind()*).

rebaseCorrelationAtTicks (*tickValue*)
Changes the *correlation* property to an equivalent correlation (that does not change the timing relationship between parent clock and this clock) where the tick value for this clock is the provided tick value.

speed

(read/write float) The speed at which the clock is running. Does not change the reported tickRate value, but will affect how ticks are calculated from parent clock ticks. Default = 1.0 = normal tick rate.

(Documentation inherited from *ClockBase*)

tickRate
Read or change the tick rate (in ticks per second) of this clock. The value read is not affected by the value of the *speed* property.

ticks
(read only) The tick count for this clock.

(Documentation inherited from *ClockBase*)

toOtherClockTicks (*otherClock*, *ticks*)
Converts a tick value for this clock into a tick value corresponding to the timescale of another clock.

Parameters

- **otherClock** – A *clock* object representing another clock.
- **ticks** – A time (tick value) for this clock

Returns The tick value of the *otherClock* that represents the same moment in time.

Throws NoCommonClock if there is no common ancestor clock (meaning it is not possible to convert

toParentTicks (*ticks*)

Method to convert from a tick value for this clock to the equivalent tick value (representing the same point in time) for the parent clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

returns The specified tick value of this clock converted to the timescale of the parent clock.

throws StopIteration if this clock has no parent

(Documentation inherited from *ClockBase*)

unbind (*dependent*)
Unbind from notification if this clock changes.

Parameters **dependent** – The dependent to unbind from receiving notifications.

TunableClock - Clock with dynamically adjustable frequency and tick offset

class dvbcss.clock.**TunableClock** (*parentClock*, *tickRate*, *ticks=0*, ***kwargs*)
A clock whose tick offset and speed can be adjusted on the fly. Must be based on another clock.

Advancement of time of this clock is based on the tick count and rates reported by the supplied parent clock.

If you adjust the tickRate or speed, then the change is applied going forward from the moment it is made. E.g. if you are observing the rate of increase of the ticks property, then doubling the speed wil cause the ticks property to start increasing faster but will not cause it to suddenly jump value.

Note: The maths to calculate and convert tick values will be performed, by default, as integer maths unless the parameters controlling the clock (tickRate etc) are floating point, or the ticks property of the parent clock supplies floating point values.

Parameters

- **parentClock** – The parent clock for this clock.
- **tickRate** – The tick rate (ticks per second) for this clock.
- **ticks** – The starting tick value for this clock.

The specified starting tick value applies from the moment this object is initialised.

adjustTicks(*offset*)
Change the tick count of this clock by the amount specified.

bind(*dependent*)
Bind for notification if this clock changes.

Parameters **dependent** – When this clock changes, the *notify()* method of this dependent will be called, passing a single argument that is this clock.

calcWhen(*ticksWhen*)

Return "when" in terms of the underlying clock behind the root clock implementation (e.g. `monotonic_time.time()` in the case of *SysClock*)

(Documentation inherited from *ClockBase*)

fromParentTicks(*ticks*)

Method to convert from a tick value for this clock's parent to the equivalent tick value (representing the same point in time) for this clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

returns The specified tick value for the parent clock converted to the timescale of this clock.

throws StopIteration if this clock has no parent

(Documentation inherited from *ClockBase*)

getEffectiveSpeed()
Returns the 'effective speed' of this clock.

This is equal to multiplying together the speed properties of this clock and all of the parents up to the root clock.

getParent()

returns *ClockBase* representing the immediate parent of this clock, or None if it is a root clock.

(Documentation inherited from *ClockBase*)

nanos
(read only) The tick count of this clock, but converted to units of nanoseconds, based on the current tick rate (but ignoring the *speed* property).

nanosToTicks(*nanos*)
Convert the supplied nanosecond to number of ticks given the current tick rate of this clock (but ignoring the *speed* property).

Parameters **nanos** – nanoseconds value

Returns number of ticks equivalent to the supplied nanosecond value

notify(*cause*)
Call to notify this clock that the clock on which it is based (its parent) has changed relative to the underlying timing source.

Parameters **cause** – The clock that is calling this method.

Will notify all dependents of this clock (entities that have registered themselves by calling *bind()*).

slew
This is an alternative method of querying or adjusting the speed property.

The slew (in ticks per second) currently applied to this clock.

Setting this property will set the speed property to correspond to the specified slew.

For example: for a clock with tickRate of 100, then a slew of -25 corresponds to a speed of 0.75

speed

(read/write float) The speed at which the clock is running. Does not change the reported tickRate value, but will affect how ticks are calculated from parent clock ticks. Default = 1.0 = normal tick rate.

(Documentation inherited from *ClockBase*)

tickRate
Read or change the tick rate (in ticks per second) of this clock. The value read is not affected by the value of the *speed* property.

ticks
(read only) The tick count for this clock.

(Documentation inherited from *ClockBase*)

toOtherClockTicks(*otherClock*, *ticks*)
Converts a tick value for this clock into a tick value corresponding to the timescale of another clock.

Parameters

- **otherClock** – A *clock* object representing another clock.
- **ticks** – A time (tick value) for this clock

Returns The tick value of the *otherClock* that represents the same moment in time.

Throws NoCommonClock if there is no common ancestor clock (meaning it is not possible to convert

toParentTicks(*ticks*)

Method to convert from a tick value for this clock to the equivalent tick value (representing the same point in time) for the parent clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

returns The specified tick value of this clock converted to the timescale of the parent clock.

throws StopIteration if this clock has no parent

(Documentation inherited from *ClockBase*)

unbind(*dependent*)
Unbind from notification if this clock changes.

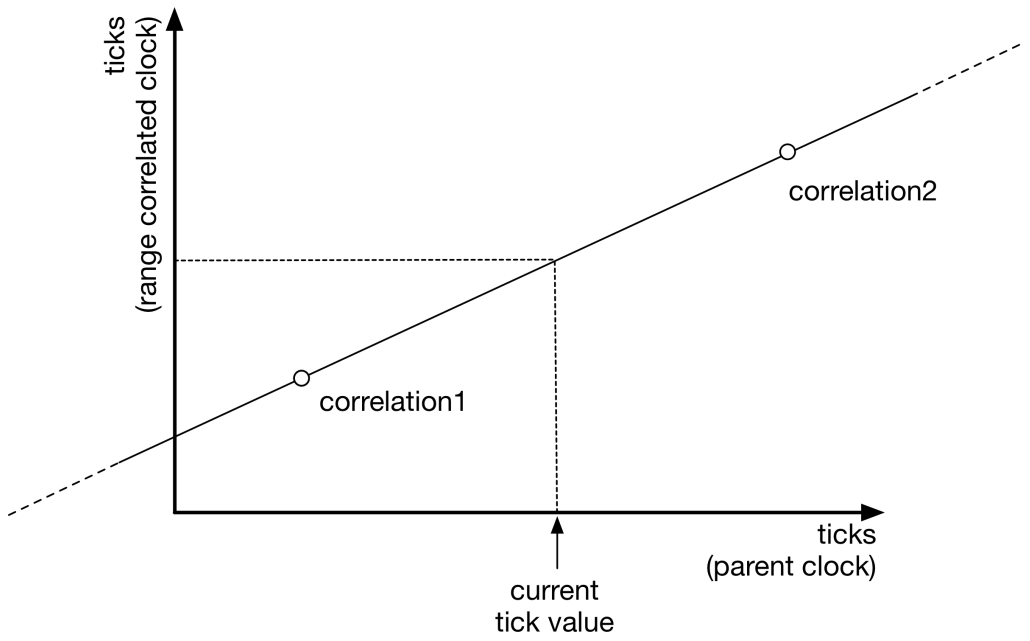
Parameters **dependent** – The dependent to unbind from receiving notifications.

RangeCorrelatedClock - Clock correlated to another clock

class dvbcss.clock.**RangeCorrelatedClock**(*parentClock*, *tickRate*, *correlation1*, *correlation2*, ***kwargs*)
A clock locked to the tick count of the parent clock by two different points of correlation.

Each correlation is a tuple *(parentTicks, selfTicks)*

This relationship can be illustrated as follows:



The tickRate you set is purely advisory - it is the tickRate reported to clocks that use this clock as the parent, and may differ from what the reality of the two correlations represents!

Parameters

- **parentClock** – The parent clock for this clock.
- **tickRate** – The advisory tick rate (ticks per second) for this clock.
- **correlation1** – (tuple(int, int)) The first point of correlation for this clock. A tuple (parent tick value, this clock tick value)
- **correlation2** – (tuple(int, int)) The second point of correlation for this clock. A tuple (parent tick value, this clock tick value)

bind(*dependent*)
Bind for notification if this clock changes.

Parameters **dependent** – When this clock changes, the *notify()* method of this dependent will be called, passing a single argument that is this clock.

calcWhen(*ticksWhen*)

Return "when" in terms of the underlying clock behind the root clock implementation (e.g. `monotonic_time.time()` in the case of *SysClock*)

(Documentation inherited from *ClockBase*)

correlation1
Read or change the first correlation tuple *(parentTicks, selfTicks)* of this clock to its parent clock.

Assign a new tuple *(parentTicks, selfTicks)* to change the correlation. This value must be a tuple, not a list.

correlation2
Read or change the first correlation tuple *(parentTicks, selfTicks)* of this clock to its parent clock.

Assign a new tuple *(parentTicks, selfTicks)* to change the correlation. This value must be a tuple, not a list.

fromParentTicks(*ticks*)

Method to convert from a tick value for this clock's parent to the equivalent tick value (representing the same point in time) for this clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

returns The specified tick value for the parent clock converted to the timescale of this clock.

throws StopIteration if this clock has no parent

(Documentation inherited from *ClockBase*)

getEffectiveSpeed()
Returns the 'effective speed' of this clock.

This is equal to multiplying together the speed properties of this clock and all of the parents up to the root clock.

getParent()

returns *ClockBase* representing the immediate parent of this clock, or None if it is a root clock.

(Documentation inherited from *ClockBase*)

nanos
(read only) The tick count of this clock, but converted to units of nanoseconds, based on the current tick rate (but ignoring the *speed* property).

nanosToTicks(*nanos*)
Convert the supplied nanosecond to number of ticks given the current tick rate of this clock (but ignoring the *speed* property).

Parameters **nanos** – nanoseconds value

Returns number of ticks equivalent to the supplied nanosecond value

notify(*cause*)
Call to notify this clock that the clock on which it is based (its parent) has changed relative to the underlying timing source.

Parameters **cause** – The clock that is calling this method.

Will notify all dependents of this clock (entities that have registered themselves by calling *bind()*).

speed

(read/write float) The speed at which the clock is running. Does not change the reported tickRate value, but will affect how ticks are calculated from parent clock ticks. Default = 1.0 = normal tick rate.

(Documentation inherited from *ClockBase*)

tickRate

Read the tick rate (in ticks per second) of this clock. The value read is not affected by the value of the *speed* property.

ticks

(read only) The tick count for this clock.

(Documentation inherited from *ClockBase*)

toOtherClockTicks(*otherClock*, *ticks*)

Converts a tick value for this clock into a tick value corresponding to the timescale of another clock.

Parameters

- **otherClock** – A *clock* object representing another clock.
- **ticks** – A time (tick value) for this clock

Returns The tick value of the *otherClock* that represents the same moment in time.

Throws NoCommonClock if there is no common ancestor clock (meaning it is not possible to convert

toParentTicks(*ticks*)

Method to convert from a tick value for this clock to the equivalent tick value (representing the same point in time) for the parent clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

returns The specified tick value of this clock converted to the timescale of the parent clock.

throws StopIteration if this clock has no parent

(Documentation inherited from *ClockBase*)

unbind(*dependent*)

Unbind from notification if this clock changes.

Parameters **dependent** – The dependent to unbind from receiving notifications.

3.3 Task scheduling for clocks

Module: *dvbcss.task*

- *Introduction*
- *Example*
- *Functions*

3.3.1 Introduction

The *dvbcss.task* module provides sleep and scheduling functions for use with the *dvbcss.clock* module. These functions track adjustments to clocks (such as changes in the tick rate or tick value/offset) to ensure that the sleep or scheduled event happen when the clock actually reaches the target tick count value.

To use this module, just import it and directly call the functions *sleepFor()*, *sleepUntil()*, *scheduleEvent()* or *runAt()*.

Note: Scheduling happens on a single thread, so if you use the *runAt()* function, try to keep the callback code as fast and simple as possible, so that it returns control as quickly as possible.

See How the dvbcss.task module works internally for information on how the internals of the Task module work.

3.3.2 Example

A simple example:

```
from dvbcss.clock import SysClock
from dvbcss.clock import CorrelatedClock
from dvbcss.task import sleepFor, runAt

s = SysClock()
c = CorrelatedClock(parentClock=s, tickRate=1000)

# wait 1 second
sleepFor(c, numTicks=1000)

# schedule callback in 5 seconds
def foo(message):
    print "Callback!", message

runAt(clock=c, whenTicks=c.ticks+5000, foo, "Tick count progressed by 5 seconds")

# ... but change the correlation to make the clock jump 1 second forward
#     causing the callback to happen one second earlier
c.correlation = (c.correlation[0], c.correlation[1] + 1000)

# ... the callback will now happen in 4 seconds time instead
```

3.3.3 Functions

dvbcss.task.**sleepUntil**(*clock*, *whenTicks*)

Sleep until the specified *clock* reaches the specified tick value.

Parameters

- **clock** – (*dvbcss.clock.ClockBase*) Clock to sleep against the ticks of.

- **whenTicks** – (int) The tick value of the clock at which this function returns.

Returns after the specified tick value is reached.

dvbcss.task.**sleepFor**(*clock*, *numTicks*)
Sleep for the number of ticks of the specified clock.

Parameters

- **clock** – (*dvbcss.clock.ClockBase*) Clock to sleep against the ticks of.
- **numTicks** – (int) The number of ticks to sleep for.

Returns after the elapsed number of ticks of the specified clock have passed.

dvbcss.task.**scheduleEvent**(*clock*, *whenTicks*, *event*)
Schedule the threading.Event to be called when the specified clock reaches (or passes) the specified tick value.

Parameters

- **clock** – (*dvbcss.clock.ClockBase*) Clock to schedule the event against
- **whenTicks** – (int) The tick value of the clock at which the event is to be triggered.
- **event** – (threading.Event) python Event object that the :method:threading.Event.set method will be called on at the scheduled time

dvbcss.task.**runAt**(*clock*, *whenTicks*, *callBack*, *args=None*, *kwargs=None*)
Call the specified callback function when the specified clock reaches (or passes) the specified tick value.

The callback happens on the single thread used within the clock scheduling system. You should avoid writing code that hogs this thread to do substantial processing.

Parameters

- **clock** – (*dvbcss.clock.ClockBase*) Clock to schedule the callback against
- **whenTicks** – (int) The tick value of the clock at which the callback is to be called.
- **callback** – (callable) Function to be called
- **args** – A list of positional arguments to be passed to the callback function when it is called.
- **kwargs** – A dict of keyword arguments to be pased to the callback function when it is called.

The *dvbcss.monotonic_time* module provides a *time()* and *sleep()* functions equivalent to those in the standard python library time module. However these are guaranteeed to be monotonic and use the highest precision time sourcecs available (depending on the host operating system).

The *dvbcss.clock* module provides high level abstractions for representing clocks and timelines and the relationships between them. The client and server implementations for the DVB-CSS protocols use these objects to represent clocks and timelines.

The:mod:*Task* module provides sleep and task scheduling functions that work with *clock* objects and allow code to be called when a clock reaches a particular tick value, even if that clock is adjusted in some way after the task is scheduled.

CHAPTER 4

Internal implementation details

4.1 Protocol server implementation details

- *CSS-WC*
  - *Overview*
  - *Classes*
- *CSS-CII and CSS-TS*
  - *Overview*
  - *Classes*

4.1.1 CSS-WC

Overview

The CSS-WC server is based on a simple generic framework for building UDP servers

Classes

4.1.2 CSS-CII and CSS-TS

Module: *dvbcss.protocol.server*

Overview

The CSS-CII and CSS-TS servers subclass the WebSocket server functionality for cherrypy implemented by ws4py in the `cherrypyserver` module.

`CIIServer` and `TSServer` both inherit from a common base implementation `WSServerTool` provided in the *dvbcss.protocol.server* module.

The Tool provides the hook into cherrypy for handling the connection request and upgrading it to a WebSocket connection, spawning an object representing the WebSocket connection and which implements the WebSocket protocols.

The base server object class is intended to manage all WebSocket connections for a particular server endpoint. It therefore provides its own customised WebScoket class that is bound to that particular server object instance.

The tool is enabled via an "on" configuration when setting up the mount point in cherrypy. The tool also expects to a "handler_cls" property set in the configuration at the mount point. This property points to a WebSocket class which can be instantiated to handle the connection.

Example usage: creating a server at "ws://<host>:80/endpoint" just using the base classes provided here:

```
import cherrypy
from ws4py.server.cherrypyserver import WebSocketPlugin
from dvbcss.protocol.server import WSServerBase, WSServerTool

# plug the tool into cherrypy as "my_server"
cherrypy.tools.my_server = WSServerTool()

WebSocketPlugin(cherrypy.engine).subscribe()

# create my server
myServer = WSServerBase()

# bind it to the URL path /endpoint in the cherrypy server
class Root(object):
    @cherrypy.expose
    def endpoint(self):
        pass

cfg = {"/endpoint": {'tools.my_server.on': True,
                'tools.my_server.handler_cls': myServer.handler
               }
      }

cherrypy.tree.mount(Root(), "/", config=cfg)

# activate cherrypy web server on port 80
cherrypy.config.update({"server.socket_port":80})
cherrypy.engine.start()
```

See documentation for `WSServerBase` for information on creating subclasses to implement specific endpoints.

Classes

class .WebSocketHandler (*WebSocket*)

This class is created and returned by the `WSServerBase._makeHandlerClass()` method and each class returned is bound to the instance of `WSServerBase` that created it.

It is intended to be provided to cherrypy as the "handler_cls" configuration parameter for the WebSocket tool. It is instantiated for every connection made.

These are subclasses of the ws4py `WebSocket` class and represent an individual WebSocket connection.

Instances of this class call through to `WSServerBase._addConnection()` and `WSServerBase._removeConnection()` and `WSServerBase._receivedMessage()` to inform the parent server of the WebSocket opening, closing and receiving messages.

classmethod isEnabled (*cls*)

Returns True if the server endpoint is enabled, otherwise False.

classmethod canAllocateConnection (*cls*)

Returns True only if the connection limit of the parent server has not yet been reached. Otherwise False.

id(*self*)

Returns A human readable connection ID

4.2 How the dvbcss.task module works internally

4.2.1 Introduction

The *dvbcss.task* module internally implements a task scheduler based around a single daemon thread with an internal priority queue.

Sleep and callback methods cause a task objet to be queued. The scheduler picks up the queued task and adds it to the priority queue and binds to the Clock so that it is notified of adjustments to the clock. When a task is added to the queue, the clock is queried to calculate the true time at which the tick count is expected to be reached by calling *dvbcss.clock.ClockBase.calcWhen()*

If a clock is adjusted the affected tasks are marked as deprecated (but remain in the priority queue) and new tasks are scheduled with a recalculated time.

4.2.2 Objects

dvbcss.task.**scheduler = <dvbcss.task._Scheduler object>**

Task scheduler. Starts an internal `threading.Thread` with `theading.Thread.daemon` set to True.

This is an internal of the Task module. For normal use you should not need to access it.

Variables

- **taskheap** – the priority queue of tasks
- **addQueue** – threadsafe queue of tasks to be added to the priority queue
- **rescheduleQueue** – thereadsafe queue of clocks that have been adjusted and therefore which need to trigger rescheduling of tasks
- **updateEvent** – `theading.Event` used to wake the scheduler thread whenever there is work pending (items added to addQueue or rescheduleQueue)
- **clock_Tasks** – mapping of clocks to takss that depend on them

Running instance of the *dvbcss.task._Scheduler*

4.2.3 Classes

class dvbcss.task.**_Scheduler**(**args*, ***kwargs*)

Task scheduler. Starts an internal `threading.Thread` with `theading.Thread.daemon` set to True.

This is an internal of the Task module. For normal use you should not need to access it.

Variables

- **taskheap** – the priority queue of tasks
- **addQueue** – threadsafe queue of tasks to be added to the priority queue
- **rescheduleQueue** – thereadsafe queue of clocks that have been adjusted and therefore which need to trigger rescheduling of tasks

- **updateEvent** – `theading.Event` used to wake the scheduler thread whenever there is work pending (items added to addQueue or rescheduleQueue)
- **clock_Tasks** – mapping of clocks to takss that depend on them

Starts the scheduler thread at initialisation.

notify(*causeClock*)
Callback entry point for when a clock is adjusted

Parameters **causeClock** – (*dvbcss.clock.ClockBase*) The clock that was adjusted and is therefore causing this notification of adjustment.

run()
Main runloop of the scheduler.

While looping:

1. Checks the queue of tasks to be added to the scheduler

   The time the task is due to be executed is calculated and used as the sort key when the task is inserted into a priority queue.

2. Checks any queued requests to reschedule tasks (due to clock adjustments)

   The existing task in the scheduler priority queue is "deprecated" And a new task is scheduled with the revised time of execution

3. checks any tasks that need to now be executed

   Dequeues them and executes them, or ignores them if they are marked as deprecated

schedule(*clock*, *whenTicks*, *callBack*, *args*, *kwargs*)
Queue up a task for scheduling

Parameters

- **clock** – (*dvbcss.clock.ClockBase*) the clock against which the task is scheduled
- **whenTicks** – (int) The tick value of the clock at which the scheduled task is to be executed
- **callback** – (func) The function (the task) that will be called at the scheduled time
- **args** – (list) List of arguments to be passed to the function when it is invoked
- **kwargs** – (dict) Dictionary of keyword arguments to be passed to the function when it is invoked

stop()
Stops the scheduler if it is running.

class dvbcss.task.**_Task**(*clock*, *whenTicks*, *callBack*, *args*, *kwargs*, *n=0*)
Representation of a scheduled task. This is an internal of the Task module. For normal use you should not need to acess it.

Initialiser

Parameters

- **clock** – (*dvbcss.clock.ClockBase*) the clock against which the task is scheduled
- **whenTicks** – (int) The tick value of the clock at which the scheduled task is to be executed
- **callback** – (func) The function (the task) that will be called at the scheduled time

- **args** – (list) List of arguments to be passed to the function when it is invoked
- **kwargs** – (dict) Dictionary of keyword arguments to be passed to the function when it is invoked
- **n** – (int) Generation count. Incremented whenever the task is based on a previous task (i.e. it is a rescheduled task)

regenerateAndDeprecate()
Sets the deleted flag of this task to True, and returns a new task the same as this one but not deleted and with the scheduled time 'when' recalculated from the clock

Here are some details on parts of the internal implementation of aspects of this library.

- modindex | Full Index

This collection of Python modules provides clients and servers for the network protocols defined in the DVB "Companion Screens and Streams" (CSS) specification ETSI 103 286 part 2. There are also supporting classes that model clocks (e.g. to represent timelines) and their inter-relationships.

Use it to build clients and servers for each of the protocols (CSS-WC, CSS-CII and CSS-TS) that mock or simulate the roles of TV Devices and Companion Screen Applications for testing and prototyping.

To use this library you need to have a working understanding of these protocols and, of course, the Python programming language.

CHAPTER 5

Getting started

1. Install, following the instructions in the README.
2. Try to Run the examples.
3. Read the docs for the DVB CSS Protocol modules.
4. Use the library in you own code

CHAPTER 6

State of implementation

This library does not currently implement the *CSS-TE* or *CSS-MRS* protocols (from the DVB specification).

There are some unit tests but these mainly only cover the calculations done within clock objects and the packing and unpacking of JSON messages.

CHAPTER 7

License and Contributing

pydvbcss is licensed as open source software under the terms of the Apache License v2.0.

See the *CONTRIBUTING* and *AUTHORS* files for information on how to contribute and who has contributed to this library.

CHAPTER 8

Contact and discuss

There is a pydvbcss google group for announcements and discussion of this library.

d

dvbcss.clock, 37
dvbcss.monotonic_time, 35
dvbcss.protocol, 7
dvbcss.protocol.cii, 11
dvbcss.protocol.client.cii, 15
dvbcss.protocol.client.ts, 23
dvbcss.protocol.client.wc, 30
dvbcss.protocol.client.wc.algorithm, 30
dvbcss.protocol.server, 57
dvbcss.protocol.server.cii, 16
dvbcss.protocol.server.ts, 23
dvbcss.protocol.server.wc, 31
dvbcss.protocol.ts, 18
dvbcss.protocol.wc, 26
dvbcss.task, 54

e

examples, 1
examples.CIIClient, 5
examples.CIIServer, 4
examples.TSClient, 6
examples.TSServer, 5
examples.TVDevice, 7
examples.WallClockClient, 4
examples.WallClockServer, 4

Symbols

.WebSocketHandler (class in dvbcss.protocol.server), 58
_Scheduler (class in dvbcss.task), 59
_Task (class in dvbcss.task), 60

A

accuracy (dvbcss.protocol.cii.TimelineOption attribute), 15
actual (dvbcss.protocol.ts.AptEptLpt attribute), 22
adjustTicks() (dvbcss.clock.TunableClock method), 50
allProperties() (dvbcss.protocol.cii.CII class method), 13
AptEptLpt (class in dvbcss.protocol.ts), 22

B

bind() (dvbcss.clock.ClockBase method), 42
bind() (dvbcss.clock.CorrelatedClock method), 47
bind() (dvbcss.clock.RangeCorrelatedClock method), 53
bind() (dvbcss.clock.SysClock method), 44
bind() (dvbcss.clock.TunableClock method), 50

C

calcWhen() (dvbcss.clock.ClockBase method), 42
calcWhen() (dvbcss.clock.CorrelatedClock method), 48
calcWhen() (dvbcss.clock.RangeCorrelatedClock method), 53
calcWhen() (dvbcss.clock.SysClock method), 44
calcWhen() (dvbcss.clock.TunableClock method), 50
canAllocateConnection() (dvbcss.protocol.server..WebSocketHandler class method), 58
Candidate (class in dvbcss.protocol.wc), 28
CII (class in dvbcss.protocol.cii), 12
ClockBase (class in dvbcss.clock), 42
combine() (dvbcss.protocol.cii.CII method), 13
contentId (dvbcss.protocol.cii.CII attribute), 13
contentIdStatus (dvbcss.protocol.cii.CII attribute), 13
contentIdStem (dvbcss.protocol.ts.SetupData attribute), 20
contentTime (dvbcss.protocol.ts.Timestamp attribute), 23
ControlTimestamp (class in dvbcss.protocol.ts), 21
copy() (dvbcss.protocol.cii.CII method), 13
copy() (dvbcss.protocol.wc.WCMessage method), 28
CorrelatedClock (class in dvbcss.clock), 46
correlation (dvbcss.clock.CorrelatedClock attribute), 48
correlation1 (dvbcss.clock.RangeCorrelatedClock attribute), 53
correlation2 (dvbcss.clock.RangeCorrelatedClock attribute), 53

D

decode() (dvbcss.protocol.cii.TimelineOption class method), 15
definedProperties() (dvbcss.protocol.cii.CII method), 13
diff() (dvbcss.protocol.cii.CII class method), 14
dvbcss.clock (module), 37
dvbcss.monotonic_time (module), 35
dvbcss.protocol (module), 7
dvbcss.protocol.cii (module), 11
dvbcss.protocol.client.cii (module), 15
dvbcss.protocol.client.ts (module), 23
dvbcss.protocol.client.wc (module), 30
dvbcss.protocol.client.wc.algorithm (module), 30
dvbcss.protocol.server (module), 57
dvbcss.protocol.server.cii (module), 16
dvbcss.protocol.server.ts (module), 23
dvbcss.protocol.server.wc (module), 31
dvbcss.protocol.ts (module), 18
dvbcss.protocol.wc (module), 26
dvbcss.task (module), 54

E

earliest (dvbcss.protocol.ts.AptEptLpt attribute), 22
encode() (dvbcss.protocol.cii.TimelineOption class method), 15
encodePrecision() (dvbcss.protocol.wc.WCMessage class method), 28
examples (module), 1
examples.CIIClient (module), 5
examples.CIIServer (module), 4
examples.TSClient (module), 6

examples.TSServer (module), 5
examples.TVDevice (module), 7
examples.WallClockClient (module), 4
examples.WallClockServer (module), 4

F

fromParentTicks() (dvbcss.clock.ClockBase method), 42
fromParentTicks() (dvbcss.clock.CorrelatedClock method), 48
fromParentTicks() (dvbcss.clock.RangeCorrelatedClock method), 53
fromParentTicks() (dvbcss.clock.SysClock method), 44
fromParentTicks() (dvbcss.clock.TunableClock method), 50

G

getEffectiveSpeed() (dvbcss.clock.ClockBase method), 43
getEffectiveSpeed() (dvbcss.clock.CorrelatedClock method), 48
getEffectiveSpeed() (dvbcss.clock.RangeCorrelatedClock method), 53
getEffectiveSpeed() (dvbcss.clock.SysClock method), 44
getEffectiveSpeed() (dvbcss.clock.TunableClock method), 50
getMaxFreqError() (dvbcss.protocol.wc.WCMessage method), 28
getParent() (dvbcss.clock.ClockBase method), 43
getParent() (dvbcss.clock.CorrelatedClock method), 48
getParent() (dvbcss.clock.RangeCorrelatedClock method), 53
getParent() (dvbcss.clock.SysClock method), 45
getParent() (dvbcss.clock.TunableClock method), 50
getPrecision() (dvbcss.protocol.wc.WCMessage method), 28

I

id() (dvbcss.protocol.server..WebSocketHandler method), 58
isEnabled() (dvbcss.protocol.server..WebSocketHandler class method), 58
isNanos (dvbcss.protocol.wc.Candidate attribute), 29

L

latest (dvbcss.protocol.ts.AptEptLpt attribute), 22

M

maxFreqError (dvbcss.protocol.wc.Candidate attribute), 30
maxFreqError (dvbcss.protocol.wc.WCMessage attribute), 27
measurePrecision() (in module dvbcss.clock), 42
mrsUrl (dvbcss.protocol.cii.CII attribute), 13
msg (dvbcss.protocol.wc.Candidate attribute), 30
msgtype (dvbcss.protocol.wc.WCMessage attribute), 27

N

nanos (dvbcss.clock.ClockBase attribute), 43
nanos (dvbcss.clock.CorrelatedClock attribute), 48
nanos (dvbcss.clock.RangeCorrelatedClock attribute), 53
nanos (dvbcss.clock.SysClock attribute), 45
nanos (dvbcss.clock.TunableClock attribute), 50
nanosToTicks() (dvbcss.clock.ClockBase method), 43
nanosToTicks() (dvbcss.clock.CorrelatedClock method), 48
nanosToTicks() (dvbcss.clock.RangeCorrelatedClock method), 53
nanosToTicks() (dvbcss.clock.SysClock method), 45
nanosToTicks() (dvbcss.clock.TunableClock method), 51
notify() (dvbcss.clock.ClockBase method), 43
notify() (dvbcss.clock.CorrelatedClock method), 48
notify() (dvbcss.clock.RangeCorrelatedClock method), 53
notify() (dvbcss.clock.SysClock method), 45
notify() (dvbcss.clock.TunableClock method), 51
notify() (dvbcss.task._Scheduler method), 60

O

offset (dvbcss.protocol.wc.Candidate attribute), 29
OMIT (in module dvbcss.protocol), 32
originalOriginate (dvbcss.protocol.wc.WCMessage attribute), 28
originateNanos (dvbcss.protocol.wc.WCMessage attribute), 27

P

pack() (dvbcss.protocol.cii.CII method), 14
pack() (dvbcss.protocol.cii.TimelineOption method), 15
pack() (dvbcss.protocol.ts.AptEptLpt method), 22
pack() (dvbcss.protocol.ts.ControlTimestamp method), 21
pack() (dvbcss.protocol.ts.SetupData method), 21
pack() (dvbcss.protocol.wc.WCMessage method), 28
precision (dvbcss.protocol.wc.Candidate attribute), 30
precision (dvbcss.protocol.wc.WCMessage attribute), 27
presentationStatus (dvbcss.protocol.cii.CII attribute), 13
private (dvbcss.protocol.cii.CII attribute), 13
private (dvbcss.protocol.cii.TimelineOption attribute), 15
private (dvbcss.protocol.ts.SetupData attribute), 21
protocolVersion (dvbcss.protocol.cii.CII attribute), 13

R

RangeCorrelatedClock (class in dvbcss.clock), 52
rebaseCorrelationAtTicks() (dvbcss.clock.CorrelatedClock method), 48

receiveNanos (dvbcss.protocol.wc.WCMessage attribute), 28
regenerateAndDeprecate() (dvbcss.task._Task method), 61
rtt (dvbcss.protocol.wc.Candidate attribute), 29
run() (dvbcss.task._Scheduler method), 60
runAt() (in module dvbcss.task), 56

S

schedule() (dvbcss.task._Scheduler method), 60
scheduleEvent() (in module dvbcss.task), 56
scheduler (in module dvbcss.task), 59
setMaxFreqError() (dvbcss.protocol.wc.WCMessage method), 28
setPrecision() (dvbcss.protocol.wc.WCMessage method), 28
SetupData (class in dvbcss.protocol.ts), 20
sleep() (in module dvbcss.monotonic_time), 37
sleepFor() (in module dvbcss.task), 56
sleepUntil() (in module dvbcss.task), 55
slew (dvbcss.clock.TunableClock attribute), 51
speed (dvbcss.clock.ClockBase attribute), 43
speed (dvbcss.clock.CorrelatedClock attribute), 49
speed (dvbcss.clock.RangeCorrelatedClock attribute), 54
speed (dvbcss.clock.SysClock attribute), 45
speed (dvbcss.clock.TunableClock attribute), 51
stop() (dvbcss.task._Scheduler method), 60
SysClock (class in dvbcss.clock), 44

T

t1 (dvbcss.protocol.wc.Candidate attribute), 29
t2 (dvbcss.protocol.wc.Candidate attribute), 29
t3 (dvbcss.protocol.wc.Candidate attribute), 29
t4 (dvbcss.protocol.wc.Candidate attribute), 29
teUrl (dvbcss.protocol.cii.CII attribute), 13
tickRate (dvbcss.clock.ClockBase attribute), 43
tickRate (dvbcss.clock.CorrelatedClock attribute), 49
tickRate (dvbcss.clock.RangeCorrelatedClock attribute), 54
tickRate (dvbcss.clock.SysClock attribute), 45
tickRate (dvbcss.clock.TunableClock attribute), 51
ticks (dvbcss.clock.ClockBase attribute), 43
ticks (dvbcss.clock.CorrelatedClock attribute), 49
ticks (dvbcss.clock.RangeCorrelatedClock attribute), 54
ticks (dvbcss.clock.SysClock attribute), 45
ticks (dvbcss.clock.TunableClock attribute), 51
time() (in module dvbcss.monotonic_time), 37
TimelineOption (class in dvbcss.protocol.cii), 14
timelines (dvbcss.protocol.cii.CII attribute), 13
timelineSelector (dvbcss.protocol.cii.TimelineOption attribute), 15
timelineSelector (dvbcss.protocol.ts.SetupData attribute), 20
timelineSpeedMultiplier (dvbcss.protocol.ts.ControlTimestamp attribute), 21
timeMicros() (in module dvbcss.monotonic_time), 37
timeNanos() (in module dvbcss.monotonic_time), 37
Timestamp (class in dvbcss.protocol.ts), 23
timestamp (dvbcss.protocol.ts.ControlTimestamp attribute), 21
toOtherClockTicks() (dvbcss.clock.ClockBase method), 43
toOtherClockTicks() (dvbcss.clock.CorrelatedClock method), 49
toOtherClockTicks() (dvbcss.clock.RangeCorrelatedClock method), 54
toOtherClockTicks() (dvbcss.clock.SysClock method), 45
toOtherClockTicks() (dvbcss.clock.TunableClock method), 51
toParentTicks() (dvbcss.clock.ClockBase method), 44
toParentTicks() (dvbcss.clock.CorrelatedClock method), 49
toParentTicks() (dvbcss.clock.RangeCorrelatedClock method), 54
toParentTicks() (dvbcss.clock.SysClock method), 45
toParentTicks() (dvbcss.clock.TunableClock method), 51
toTicks() (dvbcss.protocol.wc.Candidate method), 30
transmitNanos (dvbcss.protocol.wc.WCMessage attribute), 28
tsUrl (dvbcss.protocol.cii.CII attribute), 13
TunableClock (class in dvbcss.clock), 49
TYPE_FOLLOWUP (dvbcss.protocol.wc.WCMessage attribute), 28
TYPE_REQUEST (dvbcss.protocol.wc.WCMessage attribute), 28
TYPE_RESPONSE (dvbcss.protocol.wc.WCMessage attribute), 28
TYPE_RESPONSE_WITH_FOLLOWUP (dvbcss.protocol.wc.WCMessage attribute), 28

U

unbind() (dvbcss.clock.ClockBase method), 44
unbind() (dvbcss.clock.CorrelatedClock method), 49
unbind() (dvbcss.clock.RangeCorrelatedClock method), 54
unbind() (dvbcss.clock.SysClock method), 46
unbind() (dvbcss.clock.TunableClock method), 52
unitsPerSecond (dvbcss.protocol.cii.TimelineOption attribute), 15
unitsPerTick (dvbcss.protocol.cii.TimelineOption attribute), 15
unpack() (dvbcss.protocol.cii.CII class method), 14
unpack() (dvbcss.protocol.cii.TimelineOption class method), 15
unpack() (dvbcss.protocol.ts.AptEptLpt class method), 22

unpack() (dvbcss.protocol.ts.ControlTimestamp class method), 21
unpack() (dvbcss.protocol.ts.SetupData class method), 21
unpack() (dvbcss.protocol.wc.WCMessage class method), 28
update() (dvbcss.protocol.cii.CII method), 14

W

wallClockTime (dvbcss.protocol.ts.Timestamp attribute), 23
WCMessage (class in dvbcss.protocol.wc), 27
wcUrl (dvbcss.protocol.cii.CII attribute), 13